Naval Command,
Control and Ocean
Surveillance Center RDT&E Division San Diego, CA
92152-5001

4

Technical Document 2610
December 1993

iWarp Display Module

J. J. Symanski

Technical Document 2610
December 1993

iWarp Display Module

J. J. Symanski

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER
RDT&E DIVISION
San Diego, California 92152–5001

K. E. EVANS, CAPT, USN
Commanding Officer

R. T. SHEARER
Executive Director

ADMINISTRATIVE INFORMATION

The work documented in this report was performed by Jerry Symanski of the Technology Research and Development Branch (Code 761) within the Surveillance Department of the Naval Command, Control and Ocean Surveillance Center, Research, Development, Test, and Evaluation Division.

Released by
G. W. Byram, Head
Technology Research and
Development Branch

Under authority of
J. R. Wangler, Head
Space Systems and
Technology Division

ACKNOWLEDGMENTS

Sponsorship was provided by the HPC for "Infra-Red Focal-plane-array Sensor Processing" task of the High Performance Computing project, which is, in turn, part of the Computer Technology Block Program from the Office of Naval Research.

The author is indebted to Dr. Keith Bromley, Mr. Robert Wasilausky, and Mrs. Elizabeth Wald for their support. The author also wishes to thank Dr. Jon Webb of Carnegie-Mellon University for modifying the Adapt compiler and assembly code routines.

SM

CONTENTS

FIGURES

TABLES

INTRODUCTION

The increasing speed and precision of weapons has made high-speed processing and the interpretation of data essential to Navy missions. The only viable long-term answer to these increased processing requirements is to combine the processing speed of many processors in parallel systems. NRaD has been involved in this critical area of research since the first systems were developed in 1989. Pioneering work at Carnegie–Mellon University has lead to the Intel iWarp processor, of which NRaD has two 64–node processors. Current work in the High-Performance Computing for Infrared Sensor Processing Program (NRaD project ECB2) utilizes the iWarp processor. With the high-speed processing capability of the iWarp, we need high-speed and high-resolution display capabilities to fully exploit the potential of the iWarp hardware. The goal of this work is to provide a high-resolution real-time image display module and software for the iWarp. Software will be written to program the display to work with the high-level Adapt image processing language. This will enable researchers to program in a high-level language and evaluate sensor data processed on the iWarp in real-time.

This report will describe the highlights of the design of the display module and the software developed in the course of this work. Complete schematics for the display module are presented in Appendix A. Software for testing and running demonstrations are contained in Appendix B.

DISPLAY MODULE DESCRIPTION

The display module is a custom circuit board designed specifically for the iWarp processor. The module attaches to the external memory bus of an iWarp cell. Direct attachment to an iWarp cell will take maximum advantage of the processing power of the iWarp, the high bandwidth of the iWarp cell I/O, and the existing image processing software for the iWarp.

With software written during this development, the board generates video signals to drive a high-resolution display, with images processed within the iWarp.

The module contains 4 megabytes of VRAM which will hold images of user-determined pixel depth and size. Image data are converted to analog video signals by the Inmos G364 color video controller chip. Image sizes can range from 1024-by-1024 24-bits-per-pixel true-color images to 1-bit-per-pixel monochrome images. The user is able to choose pixel depth via software. The Inmos G364 allows many choices from 600-by-400 to 1280-by-1024 pixel image sizes. The trade-off is made depending on the depth of color and monitor used. Table 1 shows the flexibility the 4-Mb VRAM and the Inmos video controller give the user. The frame rate is limited by the rate at which the iWarp can generate and transmit image data to the VRAM.

Table 1. Image characteristics for the iWarp image display module for a 1024-x-1024 pixel display.

Image Characteristic	True-Color		Pseudocolor			Monochrome
Pixel depth (bits-per-pixel)	24	16/15	8	4	2	1
Frames stored (pixels x depth)	1	2	4	8	16	32
Frame load rate (frames per second)	4/20	8/40	16/80	32/160	64/320	128/640

Note: Image resolution is limited by the bandwidth of the Inmos video generator, which is currently 135 MHz.

Figure 1 shows the completed circuit board. Figure 2 shows the component layout. The maximum size allowable for the circuit board is 4.2 by 8.9 inches. A board this size fits over the front or rear half of a Quad Cell Board (QCB). Due to the physical constraints, the VRAMs are mounted at an angle. Heat dissipation of about 0.5 watt per package requires that the packages be mounted both top and bottom to spread out the heat and maximize the effect of the cooling air, which flows upward in the chassis. The G364 is a programmable color video controller which supports a total of seven different pixel depth-operating modes: four pseudocolor modes and three true-color modes. The pseudocolor modes have pixel depths of 2, 4, 8 bits-per-pixel. True-color modes of 15, 16, and 24 bits-per-pixel use the look-up-table for gamma correction. The Inmos G364 has a 64-bit-wide data bus to input the serial data from VRAMs, so using the 256K by 4 VRAMs requires 16 packages. Furthermore, the G364 supports the interleaving of two banks of VRAMs, for a total of 32 VRAM packages. The iWarp memory interface is also 64-bits wide but has two bits of parity on each 32-bit word. Thus there must be another 4 VRAMs to contain the parity data, which the iWarp cell computes on each 32-bit word written to memory. Control circuitry is implemented in Programmable Logic Devices (PLDs) and a number of standard integrated circuits. The red, green, and blue (RGB) analog outputs of the G364 connect to the high-resolution monitor via three coaxial cables.

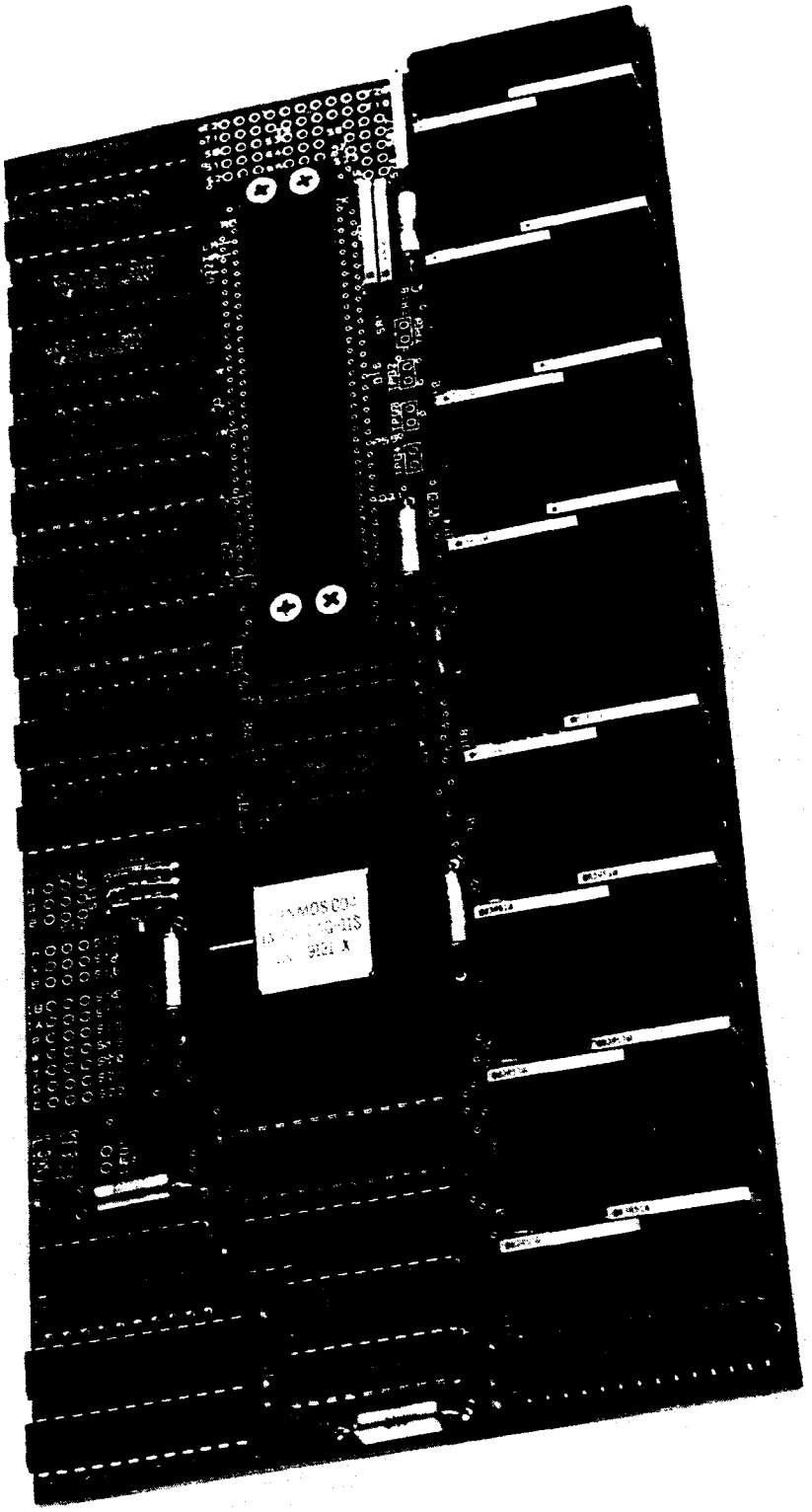


Figure 1. Display module.

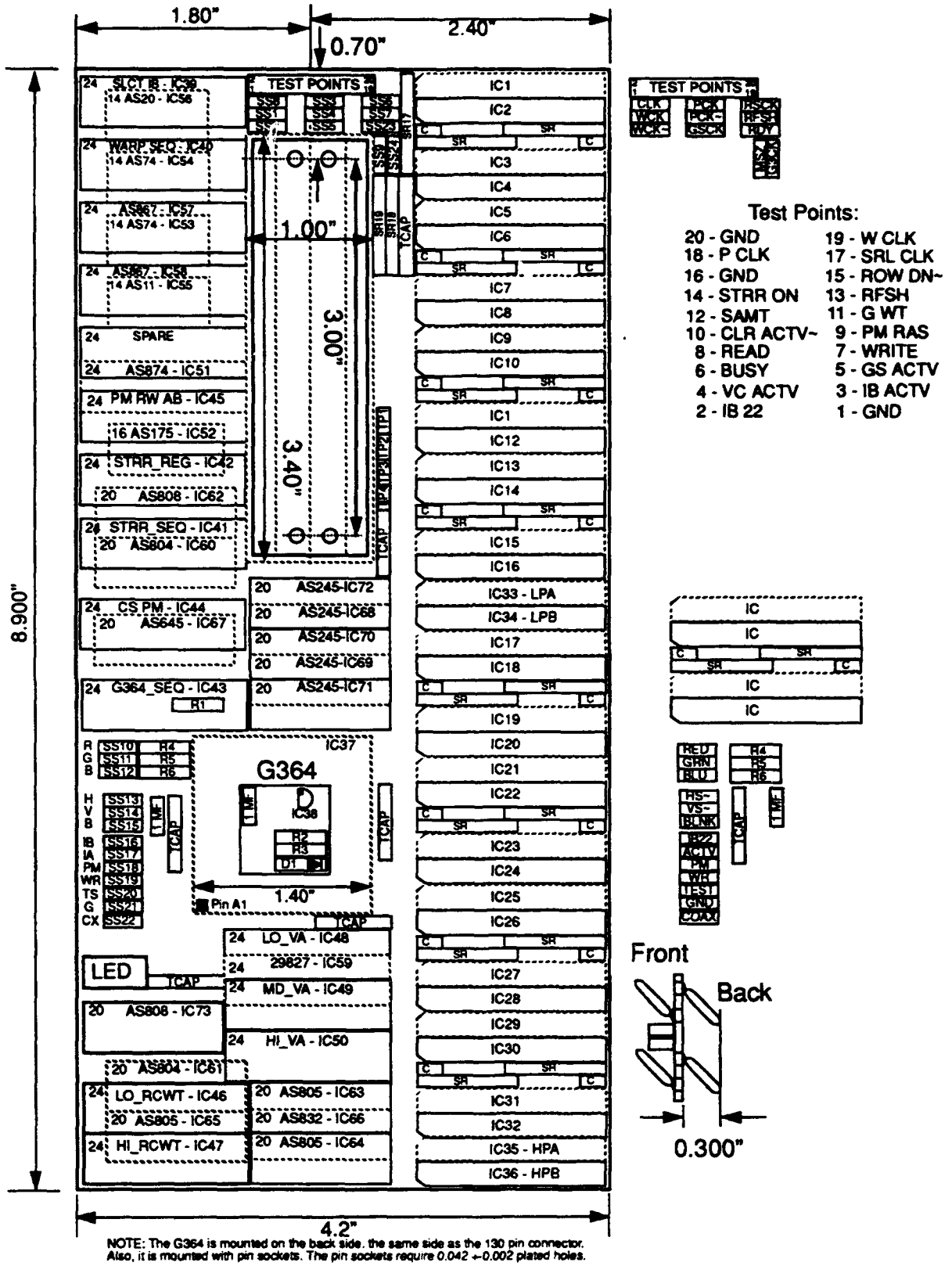


Figure 2. Layout of image board components.

BASIC OPERATION

Conceptually, the operation of the module is quite simple, due to the design of the board and the capabilities of the Inmos G364 graphics controller chip. The details of the operation of the logic will not be described here. Those interested in details should contact the author. Once installed, the module is reset with the same reset signal as the iWarp cell. The display is turned on by setting the control registers of the G364. The software to accomplish this is described in the ib.h header file in Appendix B.

Once initialized to a specific format, the module continuously displays the data in the VRAM without further intervention or control from the iWarp. The initialization is done once to start the display. The image is updated by writing into VRAM.

The display board circuitry must select between three sources, which need control of the VRAM: the iWarp, the VRAM refresh circuitry, or the G364 video controller. For a clean display, the G364 controller must be able to load a new row of data into the VRAM serial access registers at a rate dependent on the number of pixels and the pixel depth. For a 1024-by-1024 8-bits-per-pixel display, the G364 must have control of the VRAM for 2 μs every 128 μs. During this time, the iWarp and the refresh circuitry are ignored. If a request comes from the iWarp or the refresh circuitry, the request will be held until the G364 is finished, and then it will be serviced.

The VRAM can be written to in two ways: random addresses and page mode. In random addressing, any 32-bit integer (4 bytes) can be written into any of the 1,048,576 VRAM memory locations. The random write takes 650 ns since row addresses and column addresses must be given to the VRAM. The page mode write is used for the sequential writing of data. This is the usual case for an image which has been formatted into raster lines. This mode writes at about 250 ns for 8 bytes or a peak rate of 32 MB/s.

The control of the operation and the determination of the status of the display are achieved via a 4-bit control register. Control bit 0 latches in the page address for the page mode of operation. Bit 1 enables the event signal from the iWarp, which enables a quick response of the iWarp cell to the G364 requests for VRAM access. Bit 2 resets the G364 graphics chip. The reset of the G364 has special timing requirements. Bit 3 is a test bit.

As part of the development and operation of the display module, several iWarp programs were developed to aid in the use of the display. Three types of software were provided: (1) routines for the initialization of the graphics controller chip, (2) test programs for generating and displaying data on the cell with the display, and (3) Adapt high-level language routines to provide a basis upon which further image processing programs can be built.

Dr. Jon Webb at Carnegie–Mellon University has written a special version of Adapt which makes it very simple to use the display board. He has also supplied assembly code routines which minimize the time to write to the VRAM.

To the image processing application developer, data can be displayed with a simple one-line subroutine call:

```
ad_collect_image_port(out0, image_id).
```

Built-in Adapt routines gather the data from individual cells and write the data to the cell with the display board.

DESIGN RATIONALE AND APPROACH

The display module drives a high-resolution monitor capable of displaying pixel depths from 1-bit (black and white) to 24-bit true-color images, of at least 1024-by-1024 pixels. These capabilities are required to fully utilize the capabilities of the iWarp processor and achieve maximum flexibility and potential for iWarp users. The chip chosen to generate the video is the Inmos G364. This chip has a 64-bit-wide data input bus which matches the memory bus of the iWarp processor. This enables maximum data transfer to the video RAM. Another factor favoring the Inmos G364 is that it can be programmed in software to generate many display formats. The chip is simple to use. There are only the digital data input, digital control registers, and analog video output. This one chip contains circuitry to read the image data from standard VRAM, generate control signals for multiple formats, and perform sophisticated digital-to-analog signal conversion. The G364 also has 50-Ω line drivers which can be connected directly to the monitor.

The display of a 1024-by-1024 24-bit image requires 4 MB of RAM storage. At the time of the design, the 262,144-word by-4-bit VRAM was the state-of-the-art device. Availability in the plastic ZIP package made the packing of 4 MB onto the allowable circuit board possible.

PLDs are used to the largest degree possible. This is both to attain high density of logic and to ensure that even after the printed circuit board had been fabricated, there would be an ability to modify circuit operation to adjust to problems which were not anticipated.

The display module is memory mapped into the iWarp local memory address space. This provides the simplest control circuitry and the simplest functional description. There are two types of memory writing and reading. The user has the ability to read or write into random cells of the display as well as write a raster line of sequential pixels at a faster rate in page mode.

The G364 graphics chip is also memory mapped. The specific registers and values which initialize the controller for selected modes are described in the ib.h header file in Appendix B.

In order to simplify testing and minimize the impact of the initial debugging and generation of test software on iWarp users, the display module was installed in the iWarp only after the correct operation of the board was verified. A test board (figure 3) hosted by an IBM PC, was designed which emulated the hardware interface of the iWarp cell. Test images were written to the display board, using the PC so that there was a minimum of programming when the module was installed in the iWarp. This saved time and effort since the PC has a more direct interaction with the display module. The C language was used to generate the test software, so the test code was easily ported to the iWarp with only minor changes.

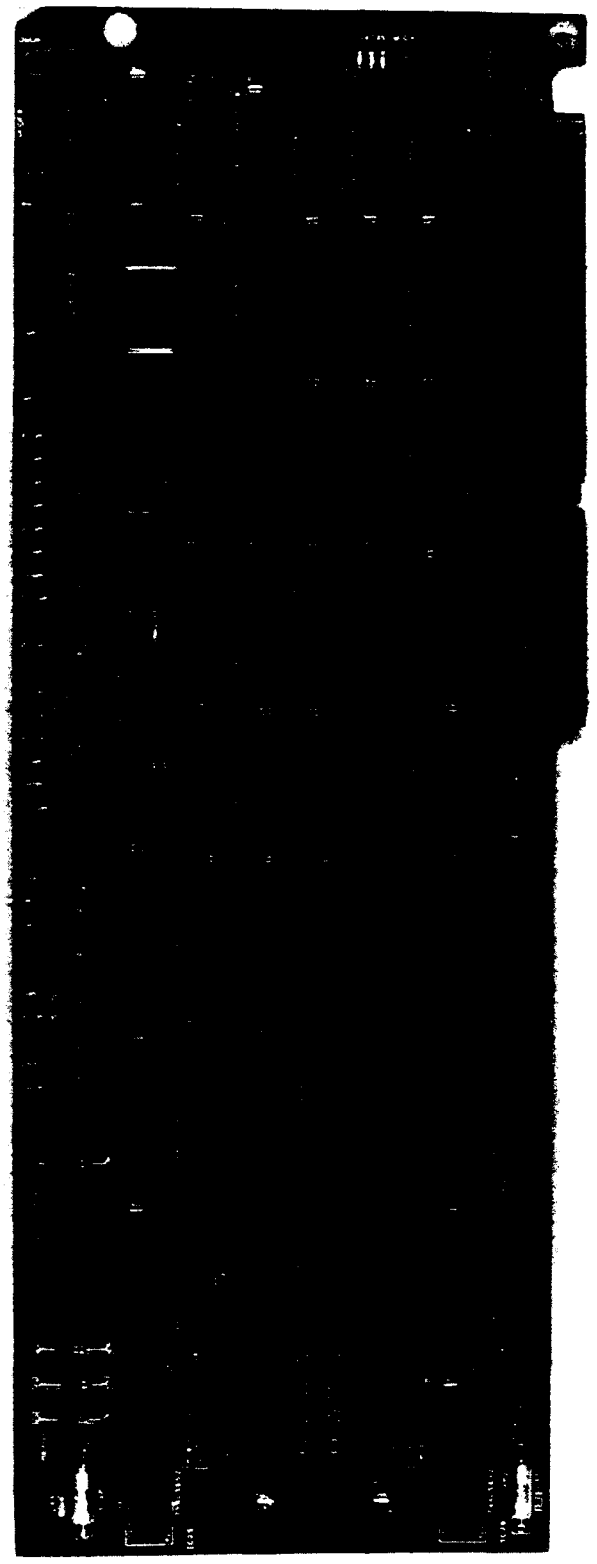


Figure 3. Test board.

DESIGN ENTRY AND SIMULATION

The complexity of a graphics board demands that we use advanced computer-aided engineering (CAE) tools to simulate and verify the operation of the circuit to the greatest extent possible, to insure the highest probability of a correct design on the first board. NRaD has a Dazix/Intergraph CAE system to perform this task. The complexity of the design makes it imperative that the design be simulated before the fabrication of the printed circuit board. The use of PLDs will incorporate a degree of flexibility into the design, even after the board has been fabricated. In any design of this size, there are unforeseen problems which only come to light after the design has progressed to the final stages.

Design entry is a highly complex and iterative process. First the components to be used in the design must be available in the CAE tools. If they are not, they must be generated in software and added to the library of available devices. The generation of models for use in simulation is a highly specialized skill in itself and can be time consuming. This design uses devices available in the Dazix design system library, with the exception of the Inmos G364 and the Toshiba Video RAM.

After components are chosen, they are connected to implement the desired logical functions using the schematic editor. Care must be taken to partition the logic in a way which will not exceed the limitations of the design system, i.e., gate counts and wiring limits. As one proceeds through the design, putting together portions of the design, simulations are run to verify that the desired operation is achieved. Careful attention must be paid to the test signals to be sure that they actually are the signals which the host system will be supplying to the board being designed. This portion of the design can generate problems if the signals described in the available documentation are inaccurate or misinterpreted.

This design was captured in 30 pages of schematics. The schematics are presented in Appendix A. Due to the complexity, the design will not be fully discussed here. Readers interested in the details of the design should contact the author directly. The first schematic shows the top level of the design. This takes three pages and shows the major blocks of the design and the connections to the iWarp cell signals as well as test points which aid in the debugging process. The control block contains nine PLDs which control the operation of the display. The logic in the PLDs is shown in corresponding schematics.

FABRICATION OF THE DISPLAY BOARD

The completed display board is shown in figure 1. This is an eight-layer printed circuit board. The net list for the board was generated by the Dazix system and converted for use on a Racal Visula system for layout. The process of generating the printed circuit board from a net list and layout schematic is a complex task in itself and was performed by engineers who have expertise in this area. The net list must be converted to the format of the layout system. Then the physical package corresponding to the components used in the logic must be taken from the layout system's library, or created, and placed on the circuit board. Physical constraints caused by the iWarp forced a high density of components. This made it necessary to try several approaches to the physical layout of the components. The final layout of the components is shown in figure 2. Components are placed on both sides of the board. Many test points are used to ease the testing process.

TEST BOARD

The test board is shown in figure 3. This board greatly eased the initial debugging of the display module by allowing a much simpler hardware and software interface for testing. Note the connector on the left side of the board. This is identical to the connector on the iWarp quad cell board onto which the module is to be installed. While the speed at which the data can be transmitted to the display is much lower with the PC than the data rate of the iWarp, the correct operation of the logic can be verified. The software developed during this initial checking of the board was also used to verify the functionality in the iWarp. The parameters necessary to set up the different modes of operation of the Inmos G364 graphics chip were determined using the test board. Only by using it, does one gain a real understanding of how a complex device such as the G364 really works. This type of knowledge is best gained in the simplest environment possible, i.e., without the complicating factors of the UNIX/Sun/iWarp software and hardware to further cloud the issues.

MONITOR REQUIREMENTS

The display module is designed to work with a high-resolution, noninterlaced monitor such as the Sony GDM–1953. The horizontal frequency of the Sony GDM–1953 is 63.34 kHz and the vertical frequency is 59.98 Hz. Resolution is 1280 by 1024. A Hitachi HM–4119 is also usable. The Inmos graphics chip is supplying red, green, and blue signals with vertical and horizontal sync signal on the green signal. The video format is composite video with plain (not tessellated) sync. There is no blanking pedestal. The interlace standard is EIA.

The parameters needed to drive the Sony monitor were derived from the Inmos G364 user manual. The parameters for several formats are documented in the header file in Appendix B.

INSTALLATION

The installation of the board into the iWarp is simple, requiring about 10 minutes. We start with a running system.

First, the iWarp must be powered down to avoid the possible crash of the host system. Change to the /iwarp/diag directory and run iwconf. When iwconf comes up enter: dep gcr=0fa. This will put the iWarp into a safe state for powering down.

Power down the iWarp. Open the chassis and remove the board to which you wish to attach the module. The module can be attached to any QCB, but some boards may be easier to work with than others. The board can mount in either the northeast or southwest corner of the QCB. However, if mounted in the southwest corner, the board will extend beyond the edge of the QCB. Thus it is best to mount the board in the northeast corner of the QCB.

The board attaches to the QCB with four Phillips head screws. Be sure to align the connector so that the screws will enter smoothly. Tighten down the screws in an "X" sequence to even out stresses.

Once the module is attached, carefully slide the QCB into the chassis. This must be done with extreme caution since the VRAMs may contact the surface mount resistors on the back of

the adjacent QCB. If there are problems, remove the QCB which the display module might bump, and slide both into the chassis together, so that there is no relative movement between the boards.

The coaxial cables which drive the monitor can be routed out the front of the chassis or through the rear and connected directly to the monitor.

TEST SOFTWARE

Appendix B contains several test programs which demonstrate the operation of the display board. These programs also act as templates as how to set up the G364 and write to VRAM in either the random or page mode. Also the event-handling code and its operation are shown.

Finally, a sample Adapt program is provided as a set of four files. The files begin with the names master.c., frame.c., frame.ad. and fastio.h. These four files are required to compile programs for Adapt. As shown in the master.c.add_one_bw program module, display of the results of image processing is accomplished with one call, namely:

```
ad_collect_image_port(out0, out_id)
```

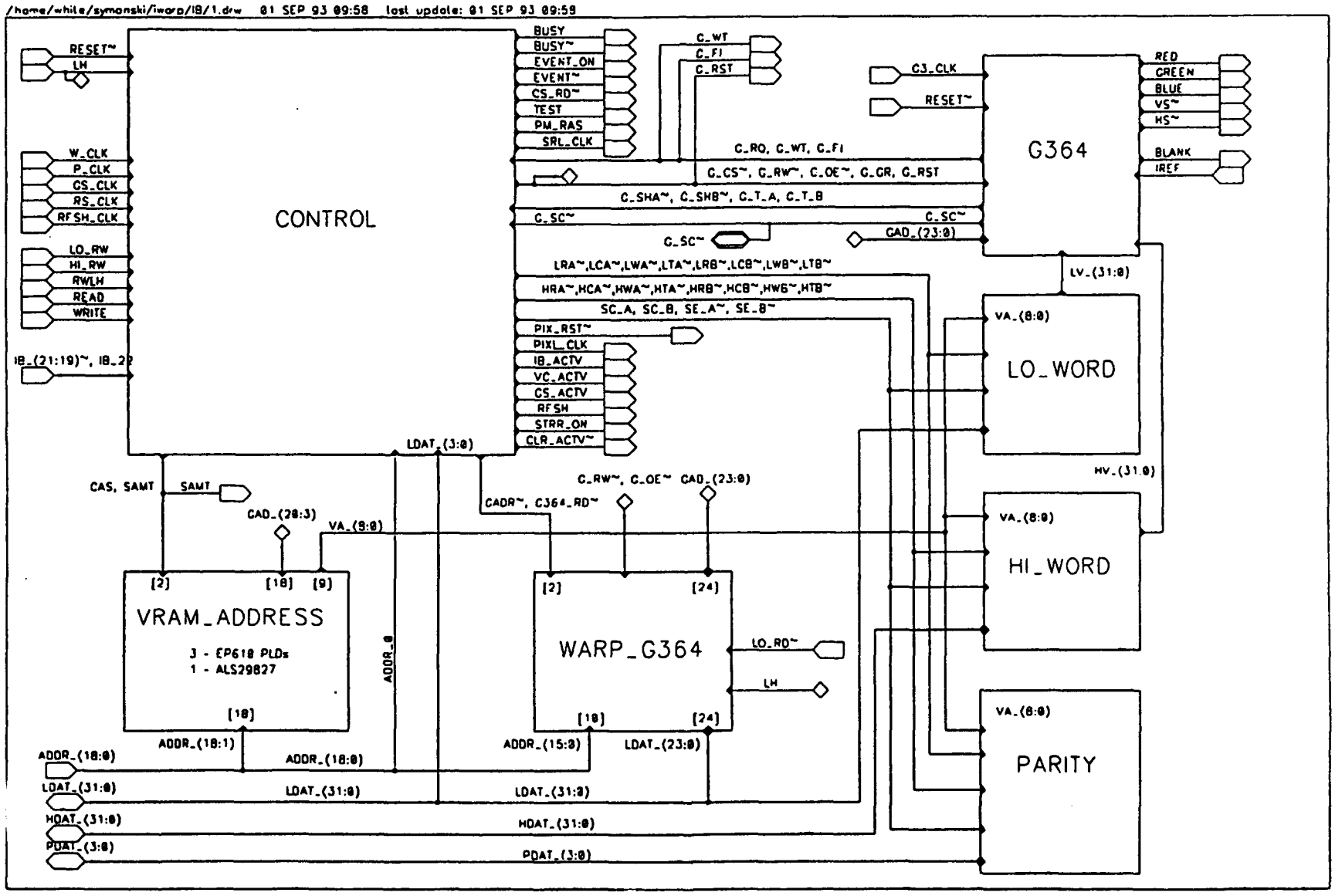
This one line is all that is necessary. The routine gathers the data from the cells, stores the image on the System Interface Board (SIB) and streams the data from the SIB directly to the display.

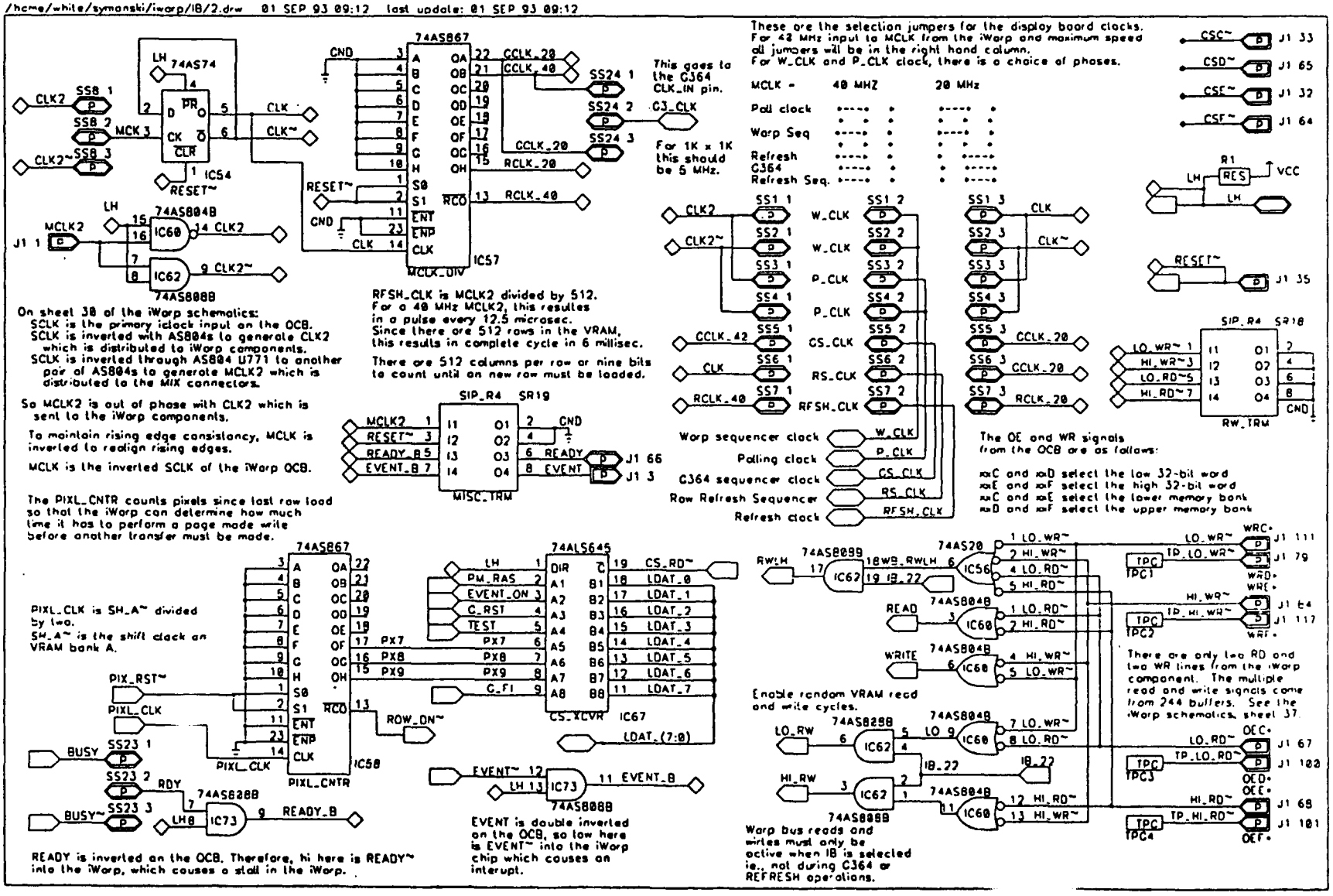
CONCLUSIONS

The design fabrication and integration of the high-resolution display module for the iWarp processor has been completed. The real-time display of images processed using the Adapt high-level programming language has been demonstrated.

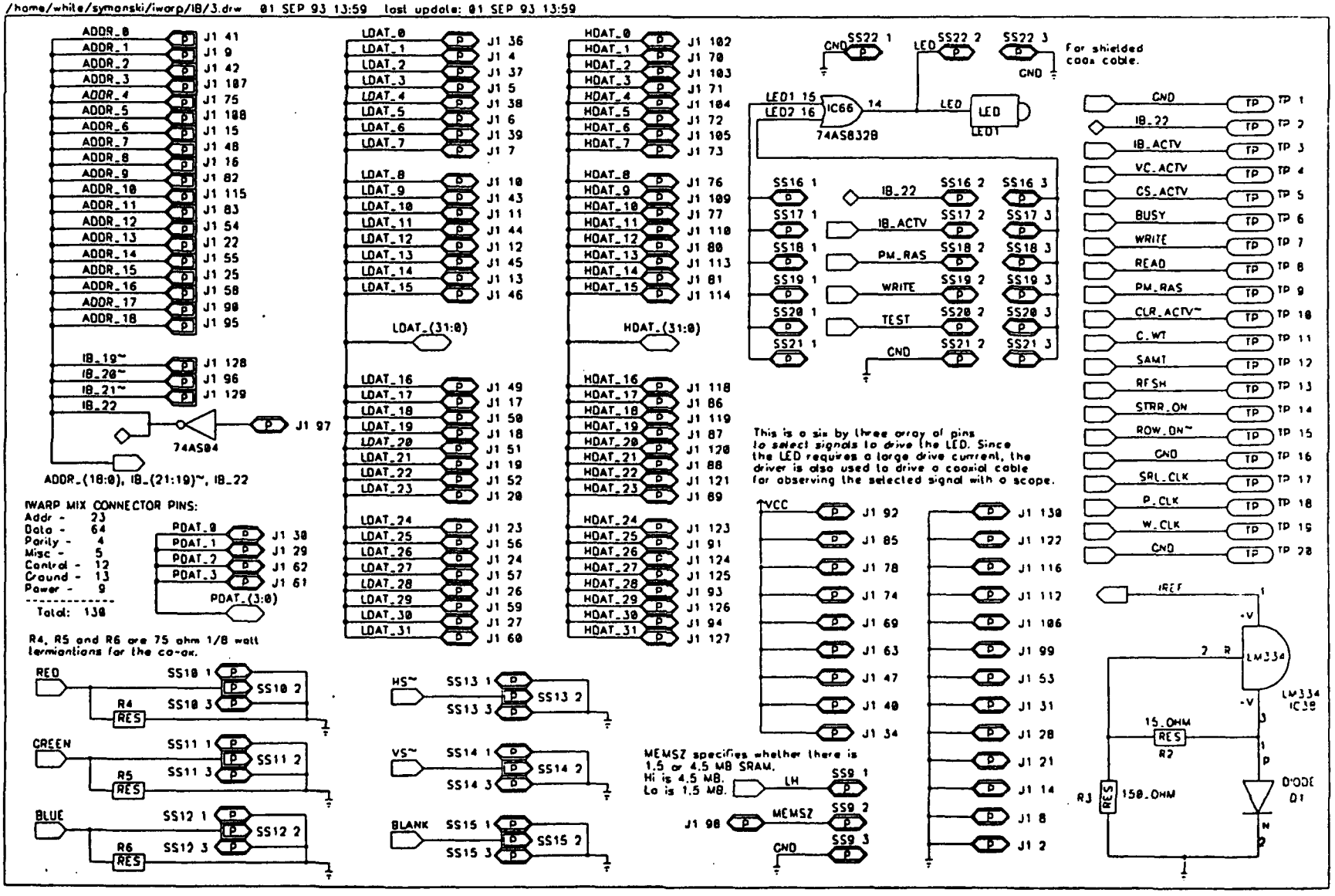
Appendix A

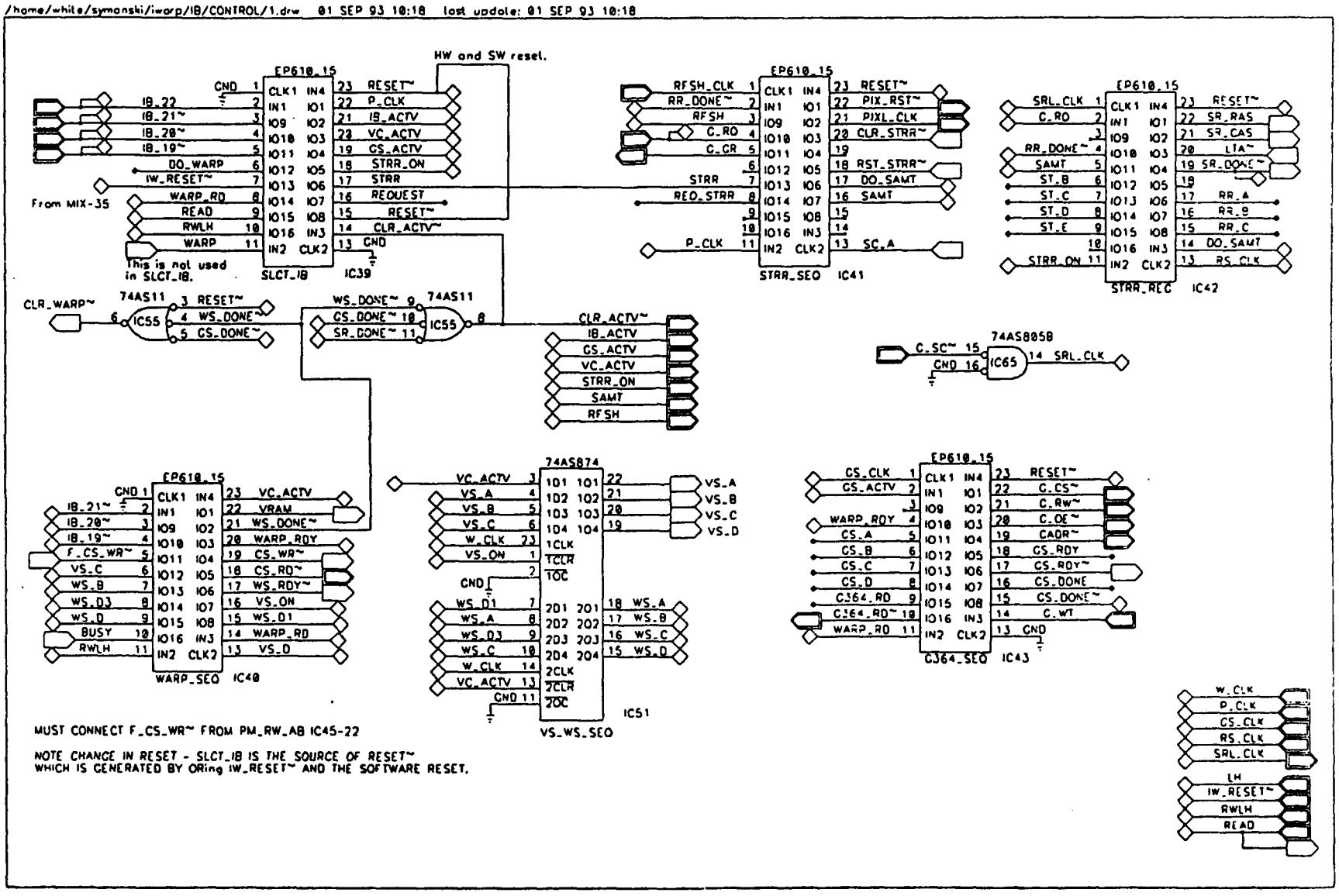
DISPLAY MODULE SCHEMATICS


/home/white/symanski/iwarp/IB/1.drw 01 SEP 93 09:58 last update: 01 SEP 93 09:59
CONTROL
RESET~
LH
W_CLK
P_CLK
CS_CLK
RS_CLK
RFSH_CLK
LO_RW
HI_RW
RWLH
READ
WRITE
IB_(21:19)~, IB_2
BUSY
BUSY~
EVENT_ON
EVENT~
CS_RD~
TEST
PM_RAS
SRL_CLK
C_WT
C_FI
C_RST
C_RQ, C_WT, C_FI
C_CS~, C_RW~, C_OE~, C_GR, C_RST
C_SHA~, C_SHB~, C_T_A, C_T_B
C_SC~
C_SC~
LRA~,LCA~,LWA~,LTA~,LRB~,LCB~,LWB~,LTB~
HRA~,HCA~,HWA~,HTA~,HRB~,HCB~,HWB~,HTB~
SC_A, SC_B, SE_A~, SE_B~
PIX_RST~
PIXL_CLK
IB_ACTV
VC_ACTV
CS_ACTV
RFSH
STRR_ON
CLR_ACTV~
LDAT_(3:0)
G364
G3_CLK
RESET~
RED
GREEN
BLUE
VS~
HS~
BLANK
IREF
GAD_(23:0)
LV_(31:0)
VA_(8:0)
LO_WORD
HV_(31:0)
HI_WORD
PARITY
CAS, SAMT
SAMT
CAD_(28:3)
VA_(9:0)
CADR~, C364_RD~
C_RW~, C_OE~ CAD_(23:0)
[2]
[18]
[9]
VRAM_ADDRESS
3 - EP610 PLDs
1 - ALS29827
ADDR_0
[2]
[24]
WARP_G364
LO_RD~
LH
[18]
[24]
ADDR_(18:1)
ADDR_(15:3)
LDAT_(23:0)
ADDR_(18:0)
LDAT_(31:0)
HDAT_(31:0)
PDAT_(3:0)

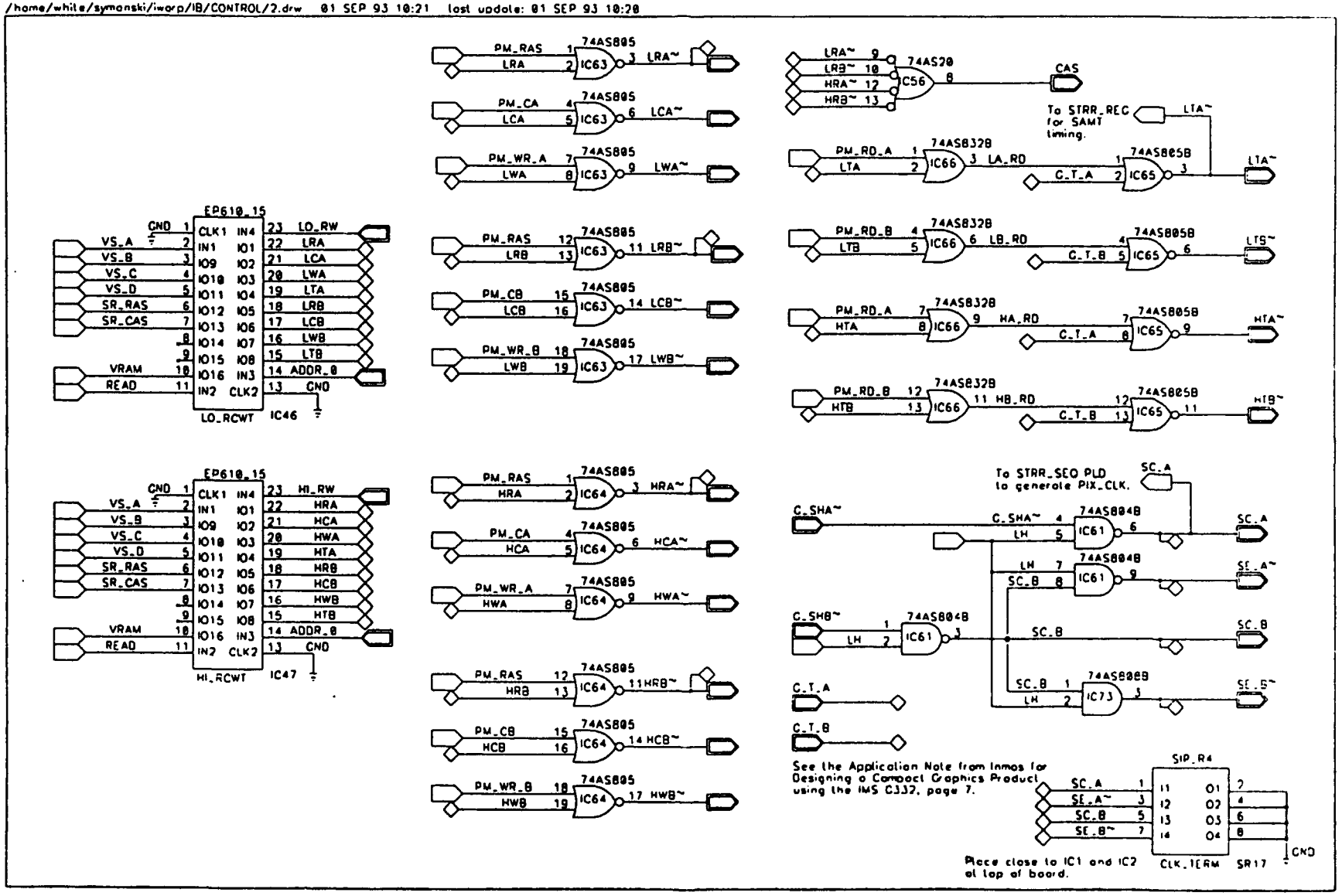

/home/white/symanski/iwarp/IB/2.drw 01 SEP 93 09:12 last update: 01 SEP 93 09:12
On sheet 38 of the iWarp schematics:
SCLK is the primary iclock input on the OCB.
SCLK is inverted with AS804s to generate CLK2 which is distributed to iWarp components.
SCLK is inverted through AS804 U771 to another pair of AS804s to generate MCLK2 which is distributed to the MIX connectors.
So MCLK2 is out of phase with CLK2 which is sent to the iWarp components.
To maintain rising edge consistency, MCLK is inverted to realign rising edges.
MCLK is the inverted SCLK of the iWarp OCB.
RFSH_CLK is MCLK2 divided by 512.
For a 40 MHz MCLK2, this results in a pulse every 12.5 microsec.
Since there are 512 rows in the VRAM, this results in complete cycle in 6 millisec.
There are 512 columns per row or nine bits to count until on new row must be loaded.
This goes to the G364 CLK_IN pin.
For 1K x 1K this should be 5 MHz.
These are the selection jumpers for the display board clocks.
For 42 MHz input to MCLK from the iWarp and maximum speed all jumpers will be in the right hand column.
For W_CLK and P_CLK clock, there is a choice of phases.
MCLK = 40 MHZ 20 MHz
Poll clock
Warp Seq
Refresh
G364
Refresh Seq.
Warp sequencer clock
Polling clock
G364 sequencer clock
Row Refresh Sequencer
Refresh clock
The OE and WR signals from the OCB are as follows:
xxC and xxD select the low 32-bit word
xxE and xxF select the high 32-bit word
xxC and xxE select the lower memory bank
xxD and xxF select the upper memory bank
The PIXL_CNTR counts pixels since last row load so that the iWarp can determine how much time it has to perform a page mode write before another transfer must be made.
PIXL_CLK is SH_A~ divided by two.
SH_A~ is the shift clock on VRAM bank A.
READY is inverted on the OCB. Therefore, hi here is READY~ into the iWarp, which causes a stall in the iWarp.
EVENT is double inverted on the OCB, so low here is EVENT~ into the iWarp chip which causes an interupt.
Enable random VRAM read and write cycles.
Warp bus reads and writes must only be active when IB is selected ie., not during G364 or REFRESH operations.
There are only two RD and two WR lines from the iWarp component. The multiple read and write signals come from 244 buffers. See the iWarp schematics, sheet 37.
74AS867
74AS804B
74AS888B
74AS74
74AS808B
74ALS645
74AS20
74AS829B
SIP_R4 SR19
SIP_R4 SR18
MISC_TRM
RW_TRM
IC54
IC57
IC58
IC60
IC62
IC67
IC73
MCLK_DIV
PIXL_CNTR
CS_XCVR
CSC~ J1 33
CSD~ J1 65
CSE~ J1 32
CSF~ J1 64
RESET~ J1 35
J1 66
J1 3
J1 111
J1 79
J1 84
J1 117
J1 67
J1 102
J1 68
J1 101

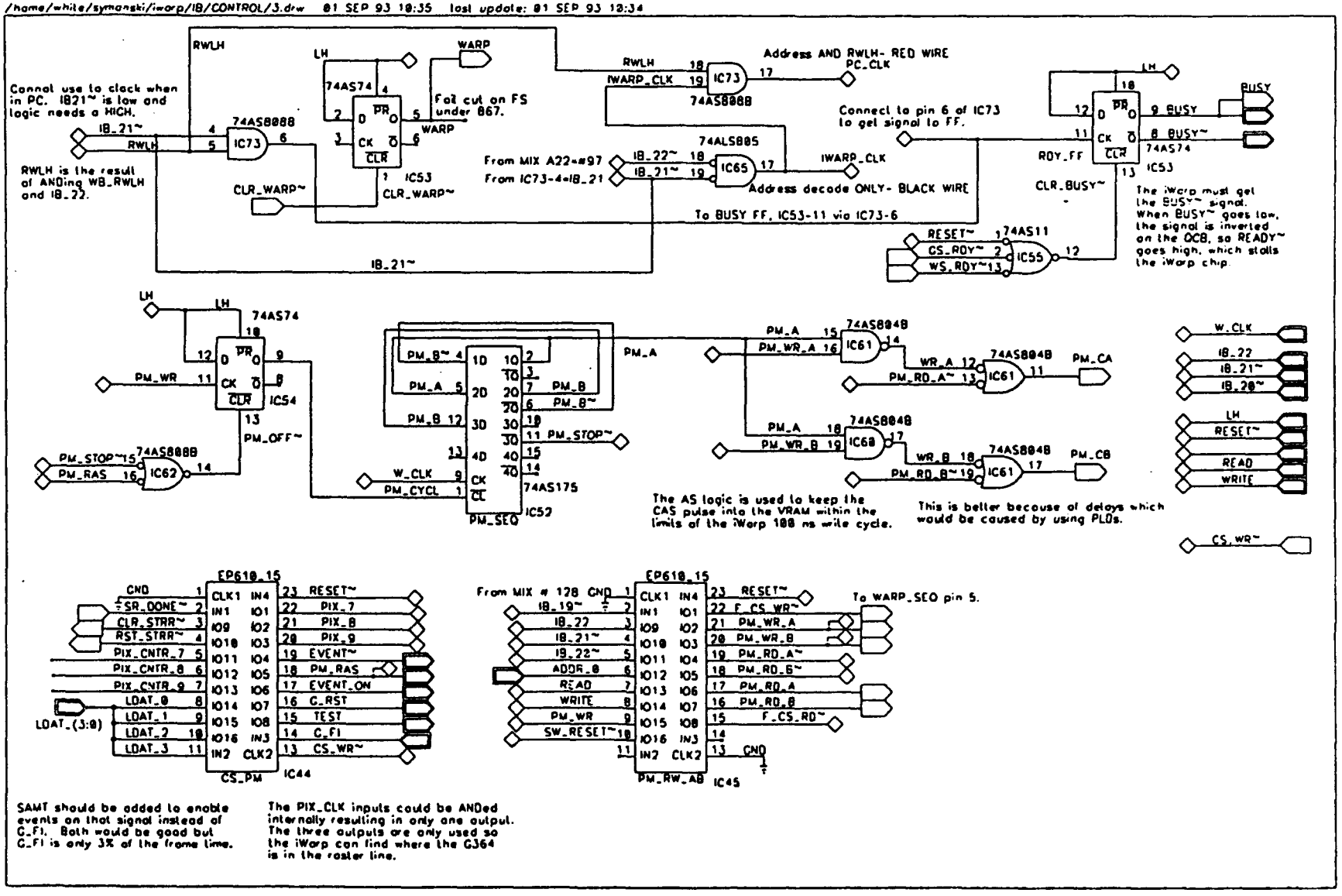
/home/white/symanski/iwarp/IB/3.drw 01 SEP 93 13:59 last update: 01 SEP 93 13:59

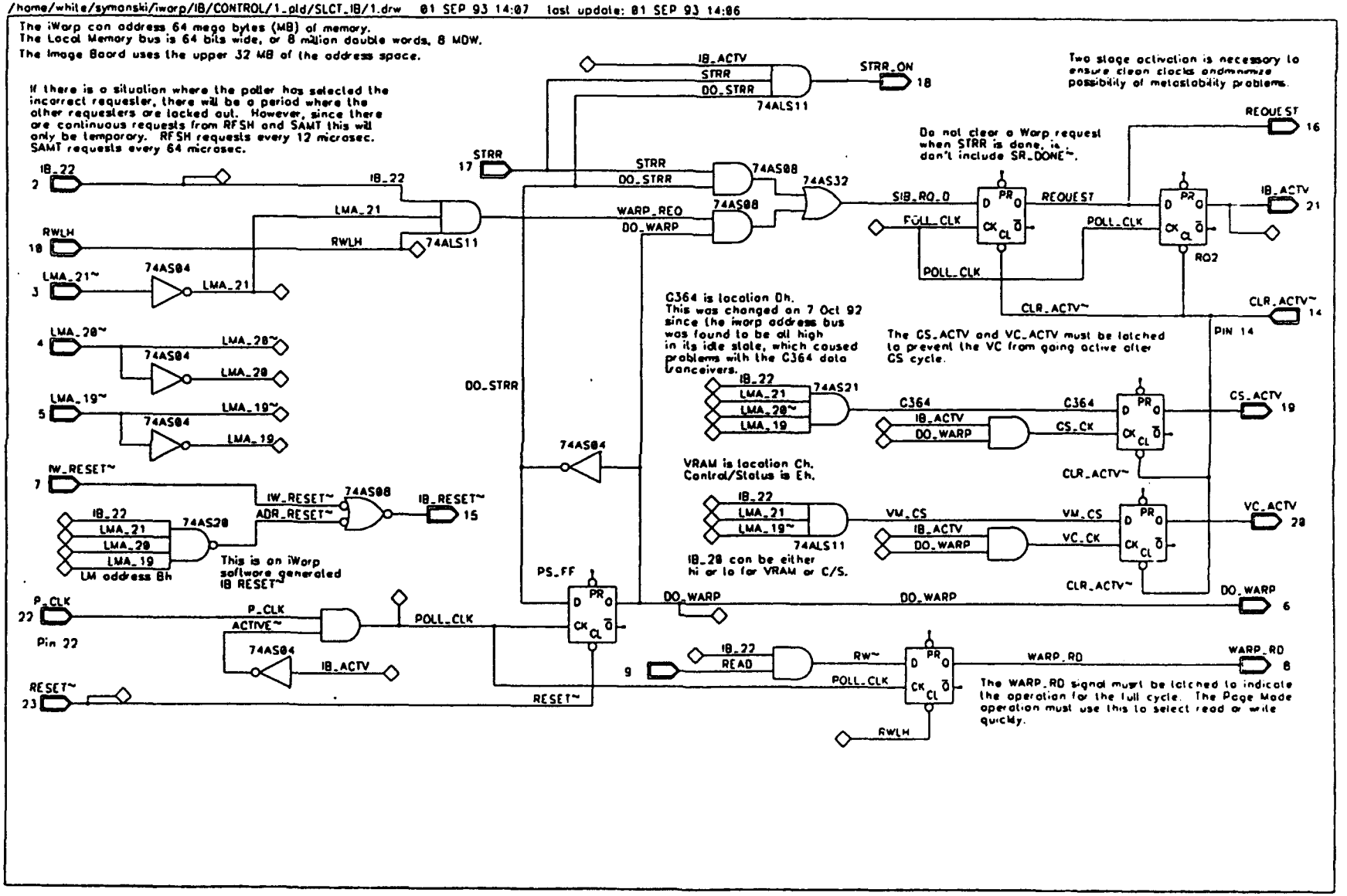



/home/white/symanski/iwarp/IB/CONTROL/1.drw 01 SEP 93 10:18 last update: 01 SEP 93 10:18
HW and SW reset.
From MIX-35
This is not used in SLCT_IB.
SLCT_IB IC39
STRR_SEQ IC41
STRR_REC IC42
WARP_SEQ IC40
VS_WS_SEQ IC51
CJ64_SEQ IC43
MUST CONNECT F_CS_WR~ FROM PM_RW_AB IC45-22
NOTE CHANGE IN RESET - SLCT_IB IS THE SOURCE OF RESET~
WHICH IS GENERATED BY ORing IW_RESET~ AND THE SOFTWARE RESET.

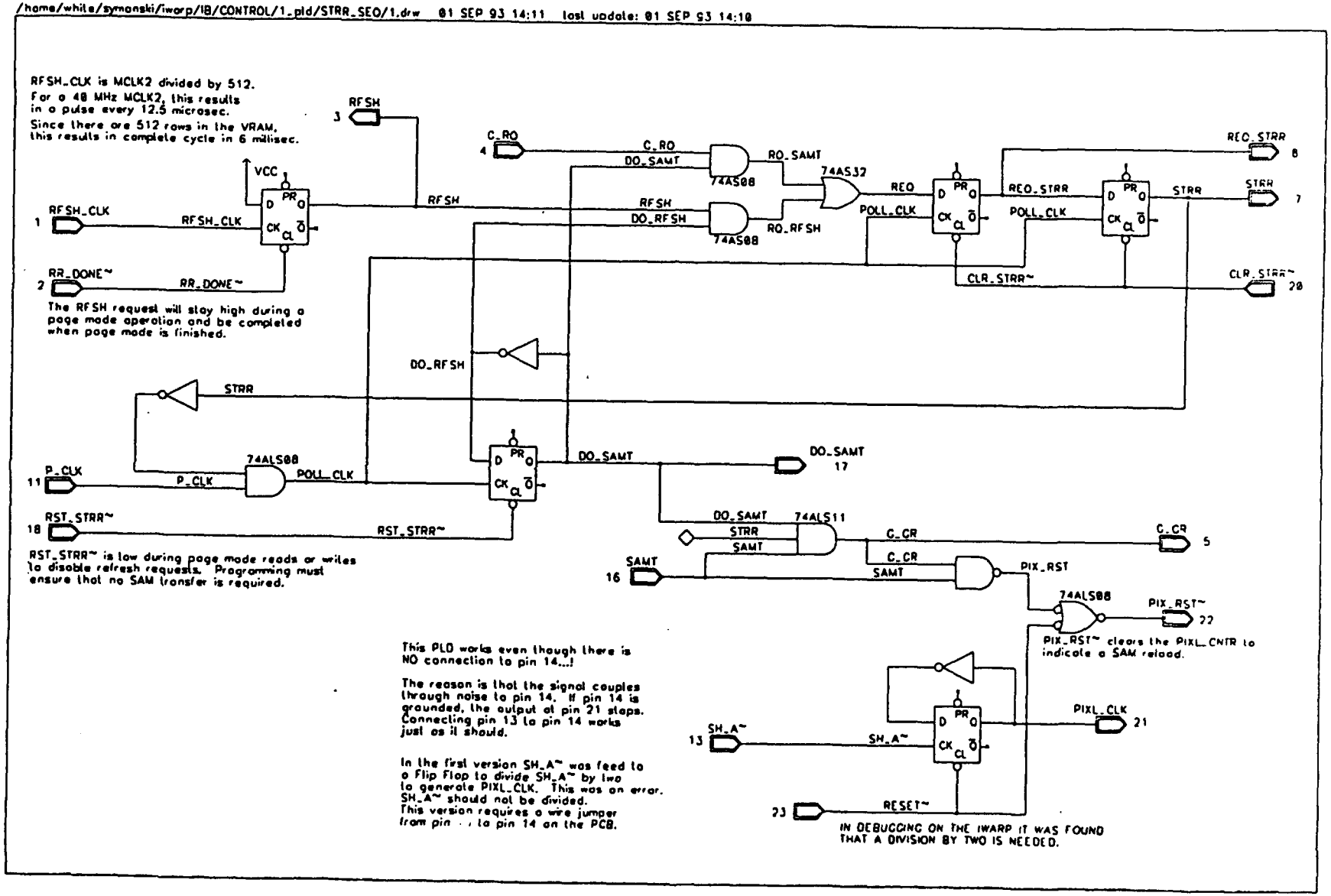
/home/white/symanski/iwarp/IB/CONTROL/2.drw 01 SEP 93 10:21 last update: 01 SEP 93 10:20

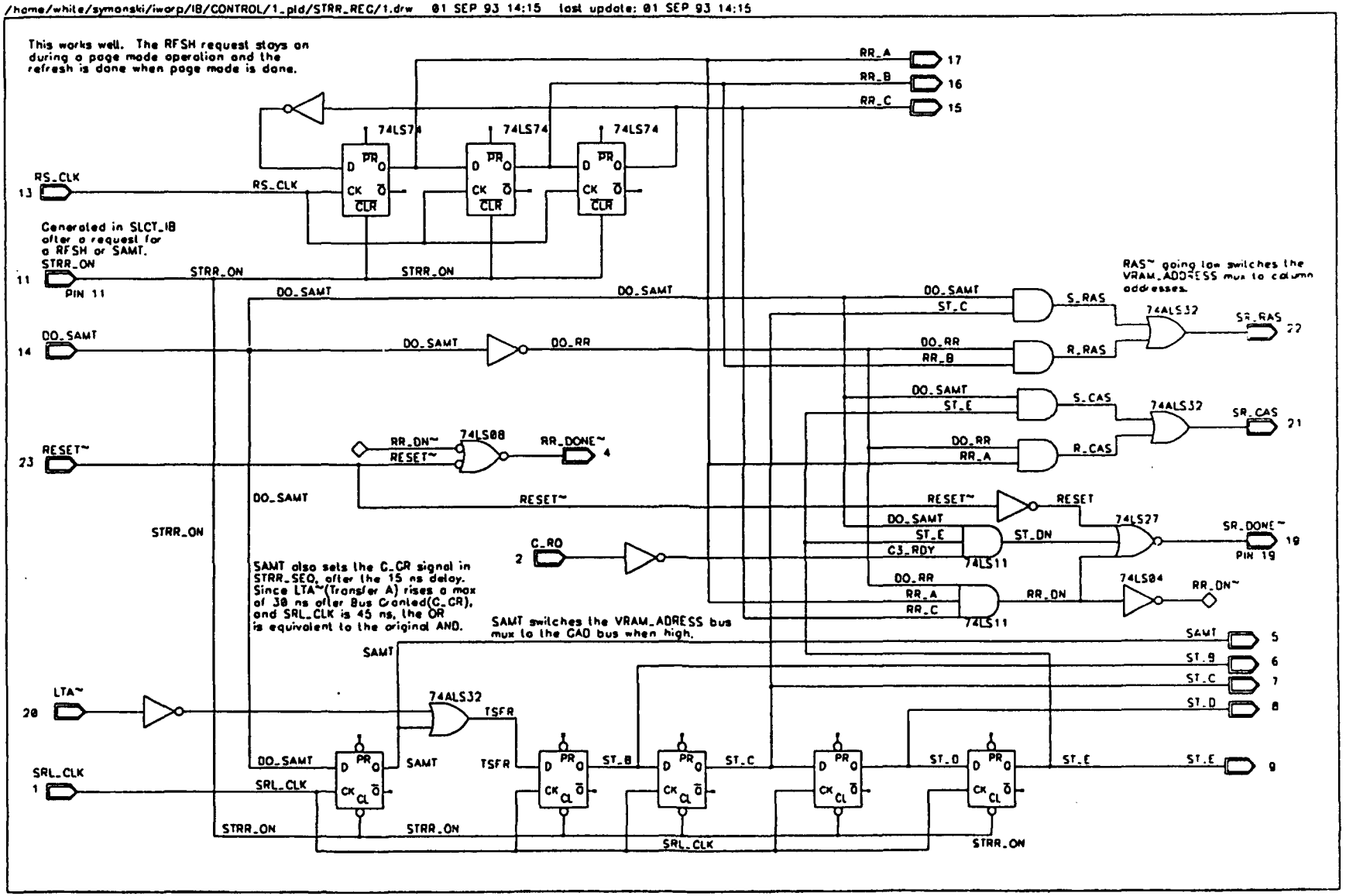


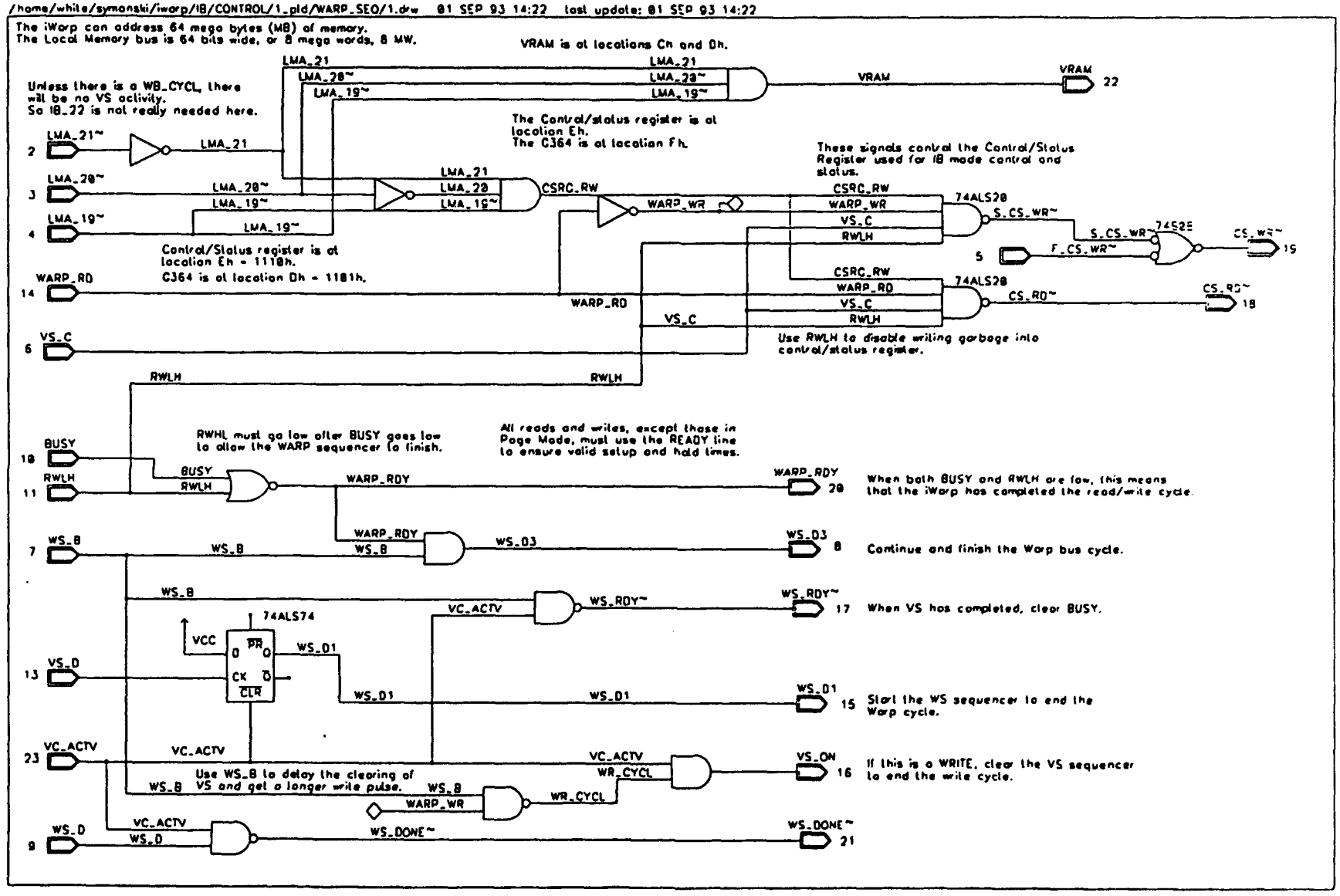

RWLH
LH
WARP
Connal use to clock when in PC. IB21~ is low and logic needs a HIGH.
IB_21~
RWLH
74AS8088
IC73
74AS74
IC53
Fal cut on FS under 867.
RWLH is the result of ANDing WB_RWLH and IB_22.
CLR_WARP~
CLR_WARP~
IB_21~
RWLH
IWARP_CLK
IC73
74AS8088
Address AND RWLH- RED WIRE PC_CLK
Connect to pin 6 of IC73 to get signal to FF.
From MIX A22-#97
From IC73-4-IB_21
IB_22~
IB_21~
74ALS805
IC65
IWARP_CLK
Address decode ONLY- BLACK WIRE
To BUSY FF, IC53-11 via IC73-6
RDY_FF
CLR_BUSY~
BUSY
9 BUSY
8 BUSY~
74AS74
IC53
The iWarp must get the BUSY~ signal. When BUSY~ goes low, the signal is inverted on the QCB, so READY~ goes high, which stalls the iWarp chip.
RESET~
CS_RDY~
WS_RDY~
74AS11
IC55
PM_WR
PM_OFF~
74AS74
IC54
PM_STOP~
PM_RAS
74AS8888
IC62
PM_B~
PM_A
PM_B
PM_STOP~
W_CLK
PM_CYCL
74AS175
IC52
PM_SEQ
PM_A
PM_WR_A
74AS8048
IC61
WR_A
PM_RD_A~
PM_CA
PM_A
PM_WR_B
IC68
WR_B
PM_RD_B~
PM_CB
The AS logic is used to keep the CAS pulse into the VRAM within the limits of the iWarp 100 ns wite cycle.
This is better because of delays which would be caused by using PLDs.
W_CLK
IB_22
IB_21~
IB_20~
LH
RESET~
READ
WRITE
CS_WR~
EP610_15
GND
SR_DONE~
CLR_STRR~
RST_STRR~
PIX_CNTR_7
PIX_CNTR_8
PIX_CNTR_9
LOAT_0
LOAT_1
LOAT_2
LOAT_3
LOAT_(3:0)
RESET~
PIX_7
PIX_8
PIX_9
EVENT~
PM_RAS
EVENT_ON
G_RST
TEST
C_FI
CS_WR~
CS_PM
IC44
From MIX # 128 GND
IB_19~
IB_22
IB_21~
IB_22~
ADDR_0
READ
WRITE
PM_WR
SW_RESET~
F_CS_WR~
PM_WR_A
PM_WR_B
PM_RD_A~
PM_RD_B~
PM_RD_A
PM_RD_B
F_CS_RD~
To WARP_SEQ pin 5.
PM_RW_AB
IC45
SAMT should be added to enable events on that signal instead of C_FI. Both would be good but C_FI is only 3% of the frame time.
The PIX_CLK inputs could be ANDed internally resulting in only one output. The three outputs are only used so the iWarp can find where the G364 is in the raster line.

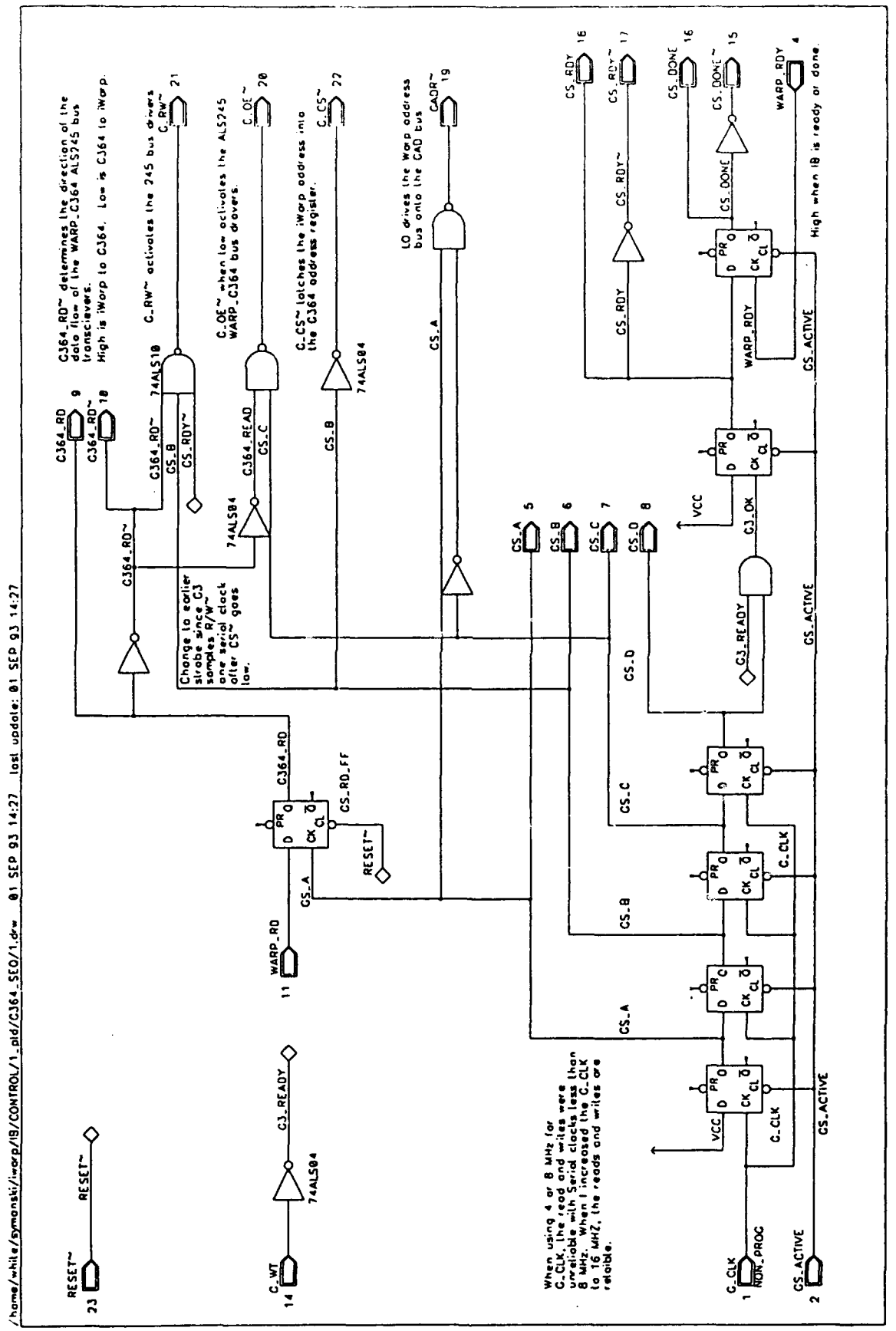

/home/white/symanski/iwarp/IB/CONTROL/1_old/SLCT_IB/1.drw 01 SEP 93 14:07 last update: 01 SEP 93 14:06
The iWarp can address 64 mega bytes (MB) of memory.
The Local Memory bus is 64 bits wide, or 8 million double words, 8 MDW.
The Image Board uses the upper 32 MB of the address space.
If there is a situation where the poller has selected the incorrect requester, there will be a period where the other requesters are locked out. However, since there are continuous requests from RFSH and SAMT this will only be temporary. RFSH requests every 12 microsec. SAMT requests every 64 microsec.
IB_ACTV
STRR
DO_STRR
74ALS11
STRR_ON
18
Two stage activation is necessary to ensure clean clocks and minimize possibility of metastability problems.
REQUEST
16
Do not clear a Warp request when STRR is done, ie. don't include SR_DONE~.
STRR
17
74AS08
74AS32
SIB_RQ_0
REQUEST
IB_ACTV
21
POLL_CLK
RQ2
CLR_ACTV~
14
PIN 14
IB_22
2
LMA_21
RWLH
18
74ALS11
WARP_REQ
DO_WARP
LMA_21~
3
74AS04
LMA_21
LMA_20~
4
LMA_20
LMA_19~
5
LMA_19
C364 is location Dh. This was changed on 7 Oct 92 since the iwarp address bus was found to be all high in its idle state, which caused problems with the C364 data tranceivers.
The CS_ACTV and VC_ACTV must be latched to prevent the VC from going active after CS cycle.
74AS21
C364
CS_ACTV
19
CS_CK
VRAM is location Ch. Control/Status is Eh.
IW_RESET~
7
IB_RESET~
15
ADR_RESET~
74AS28
LM address 8h
This is an iWarp software generated IB RESET~
IB_28 can be either hi or lo for VRAM or C/S.
VM_CS
VC_ACTV
20
VC_CK
PS_FF
P_CLK
22
Pin 22
ACTIVE~
POLL_CLK
DO_WARP
6
READ
RW~
WARP_RD
8
The WARP_RD signal must be latched to indicate the operation for the full cycle. The Page Mode operation must use this to select read or write quickly.
RESET~
23
9
RWLH

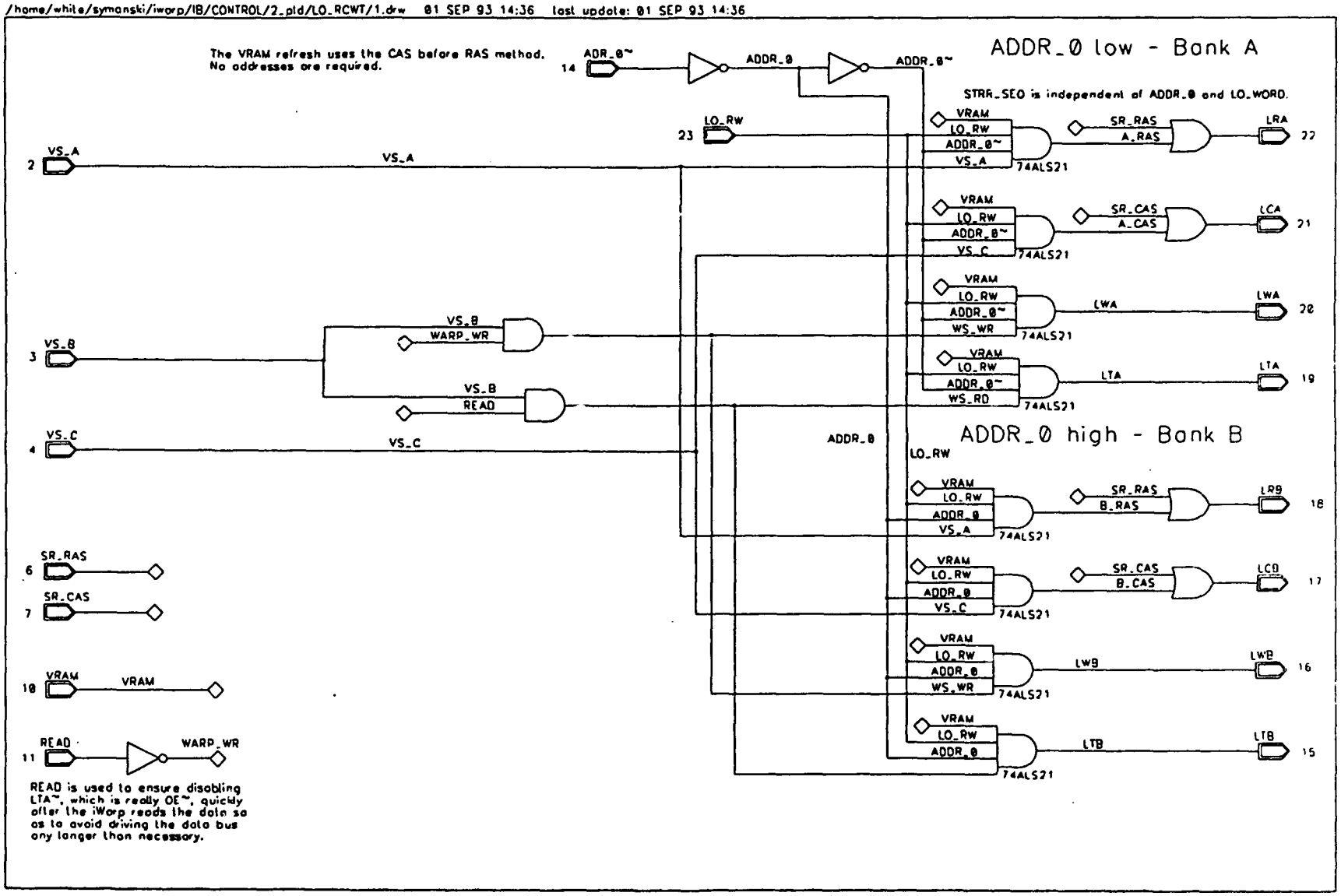
/home/white/symanski/iwarp/IB/CONTROL/1_pld/STRR_SEQ/1.drw 01 SEP 93 14:11 last update: 01 SEP 93 14:10

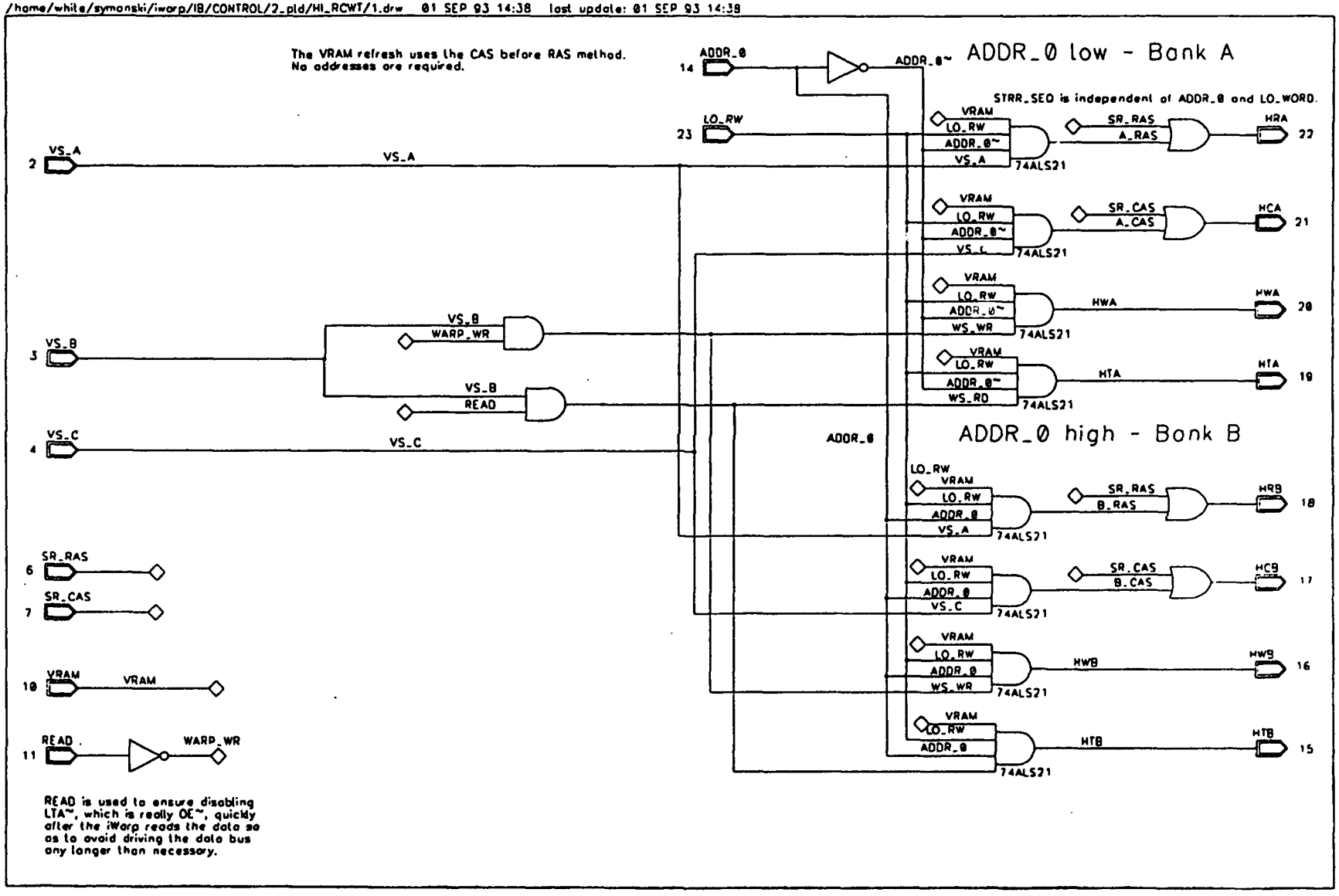



/home/white/symonski/iwarp/IB/CONTROL/1_pld/STRR_REC/1.drw 01 SEP 93 14:15 last update: 01 SEP 93 14:15
This works well. The RFSH request stays on during a page mode operation and the refresh is done when page mode is done.
RR_A
RR_B
RR_C
17
16
15
74LS74
74LS74
74LS74
RS_CLK
13
RS_CLK
Generated in SLCT_IB after a request for a RFSH or SAMT.
STRR_ON
11
PIN 11
STRR_ON
STRR_ON
DO_SAMT
DO_SAMT
DO_SAMT
14
DO_SAMT
DO_RR
RAS~ going low switches the VRAM_ADDRESS mux to column addresses.
DO_SAMT
ST_C
S_RAS
74ALS32
SR_RAS
22
DO_RR
RR_B
R_RAS
DO_SAMT
ST_E
S_CAS
74ALS32
SR_CAS
21
DO_RR
RR_A
R_CAS
RR_ON~
RESET~
74LS08
RR_DONE~
4
RESET~
23
DO_SAMT
RESET~
RESET~
RESET
DO_SAMT
ST_E
ST_DN
74LS27
SR_DONE~
19
PIN 19
74LS11
STRR_ON
C_RQ
2
C3_RDY
SAMT also sets the C_CR signal in STRR_SEQ, after the 15 ns delay. Since LTA~(Transfer A) rises a max of 30 ns after Bus Granted(C_CR), and SRL_CLK is 45 ns, the OR is equivalent to the original AND.
DO_RR
RR_A
RR_C
74LS11
RR_ON
74LS04
RR_ON~
SAMT switches the VRAM_ADRESS bus mux to the CAD bus when high.
SAMT
5
ST_B
6
ST_C
7
ST_D
8
LTA~
20
74ALS32
TSFR
SRL_CLK
1
DO_SAMT
SAMT
TSFR
ST_B
ST_C
ST_D
ST_E
ST_E
9
SRL_CLK
STRR_ON
STRR_ON
SRL_CLK
STRR_ON

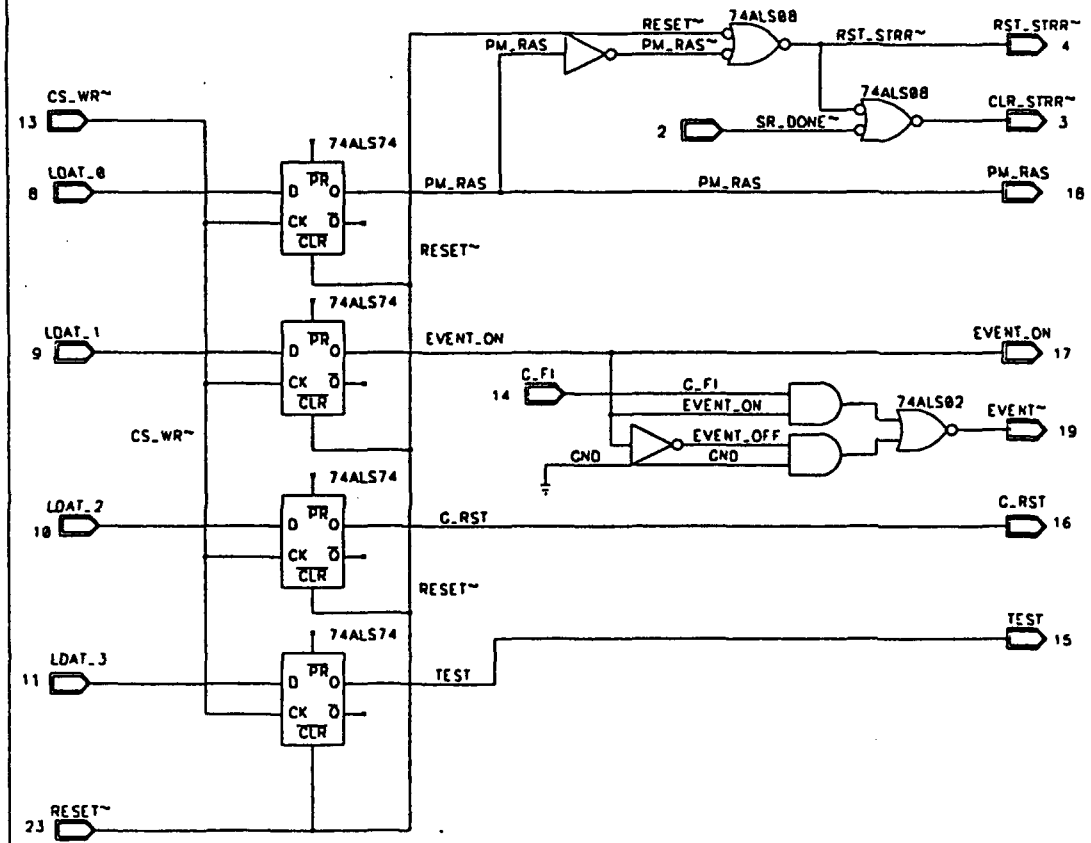

The iWarp can address 64 mega bytes (MB) of memory.
The Local Memory bus is 64 bits wide, or 8 mega words, 8 MW.
VRAM is at locations Ch and Dh.
Unless there is a WB_CYCL, there will be no VS activity.
So IB_22 is not really needed here.
The Control/status register is at location Eh.
The CJ64 is at location Fh.
These signals control the Control/Status Register used for IB mode control and status.
Control/Status register is at location Eh = 1110b.
CJ64 is at location Dh = 1101b.
Use RWLH to disable writing garbage into control/status register.
RWHL must go low after BUSY goes low to allow the WARP sequencer to finish.
All reads and writes, except those in Page Mode, must use the READY line to ensure valid setup and hold times.
When both BUSY and RWLH are low, this means that the iWarp has completed the read/write cycle.
Continue and finish the Warp bus cycle.
When VS has completed, clear BUSY.
Start the WS sequencer to end the Warp cycle.
If this is a WRITE, clear the VS sequencer to end the write cycle.
Use WS_B to delay the clearing of VS and get a longer write pulse.
74ALS74
74ALS20
74ALS08
LMA_21
LMA_20~
LMA_19~
VRAM
CSRC_RW
WARP_WR
VS_C
RWLH
S_CS_WR~
F_CS_WR~
CS_WR~
WARP_RD
CS_RD~
BUSY
WARP_RDY
WS_B
WS_D3
WS_RDY~
VCC
WS_D1
VS_D
VC_ACTV
WR_CYCL
VS_ON
WS_DONE~


/home/white/symonski/iwarp/19/CONTROL/1_pld/C364_SEQ/1.drw 01 SEP 93 14:27 last update: 01 SEP 93 14:27
C364_RD~ determines the direction of the data flow of the WARP_C364 ALS245 bus transcievers.
High is iWarp to C364. Low is C364 to iWarp.
C_RW~ activates the 245 bus drivers
C_OE~ when low activates the ALS245 WARP_C364 bus drivers.
C_CS~ latches the iWarp address into the C364 address register.
LO drives the Warp address bus onto the CAD bus
Change to earlier strobe since C3 samples R/W~ one serial clock after CS~ goes low.
When using 4 or 8 MHz for C_CLK, the read and writes were unreliable with Serial clocks less than 8 MHz. When I increased the C_CLK to 16 MHZ, the reads and writes are reliable.
High when 18 is ready or done.
RESET~
C_WT
C3_READY
74ALS04
74ALS10
C364_RD
C364_RD~
C364_READ
CS_RD_FF
WARP_RD
C_CLK
NON_PROC
CS_ACTIVE
CS_RDY
CS_RDY~
CS_DONE
CS_DONE~
WARP_RDY
C3_OK
C3_READY
VCC
CS_A
CS_B
CS_C
CS_D
CADR~


/home/white/symanski/iwarp/IB/CONTROL/2_pld/LO_RCWT/1.drw 01 SEP 93 14:36 last update: 01 SEP 93 14:36
The VRAM refresh uses the CAS before RAS method. No addresses are required.
ADDR_0~
14
ADDR_0
ADDR_0~
ADDR_0 low - Bank A
STRB_SEO is independent of ADDR_0 and LO_WORD.
LO_RW
23
VS_A
2
VS_A
VRAM
LO_RW
ADDR_0~
VS_A
74ALS21
SR_RAS
A_RAS
LRA
22
VRAM
LO_RW
ADDR_0~
VS_C
74ALS21
SR_CAS
A_CAS
LCA
21
VS_B
3
VS_B
WARP_WR
VRAM
LO_RW
ADDR_0~
WS_WR
74ALS21
LWA
LWA
20
VS_B
READ
VRAM
LO_RW
ADDR_0~
WS_RD
74ALS21
LTA
LTA
19
VS_C
4
VS_C
ADDR_0
ADDR_0 high - Bank B
LO_RW
VRAM
LO_RW
ADDR_0
VS_A
74ALS21
SR_RAS
B_RAS
LRB
18
SR_RAS
6
SR_CAS
7
VRAM
LO_RW
ADDR_0
VS_C
74ALS21
SR_CAS
B_CAS
LCB
17
VRAM
10
VRAM
VRAM
LO_RW
ADDR_0
WS_WR
74ALS21
LWB
LWB
16
READ
11
WARP_WR
VRAM
LO_RW
ADDR_0
74ALS21
LTB
LTB
15
READ is used to ensure disabling LTA~, which is really OE~, quickly after the iWarp reads the data so as to avoid driving the data bus any longer than necessary.


/home/white/symonski/iwarp/IB/CONTROL/2_pld/HI_RCWT/1.drw 01 SEP 93 14:38 last update: 01 SEP 93 14:38
The VRAM refresh uses the CAS before RAS method.
No addresses are required.
ADDR_0 low - Bank A
ADDR_0 high - Bank B
STRR_SEQ is independent of ADDR_0 and LO_WORD.
READ is used to ensure disabling LTA~, which is really OE~, quickly after the iWarp reads the data so as to avoid driving the data bus any longer than necessary.
74ALS21

/home/while/symonski/iwarp/IB/CONTROL/3_pld/CS_PM/1.drw 31 SEP 93 10:57 last update: 01 SEP 93 10:57

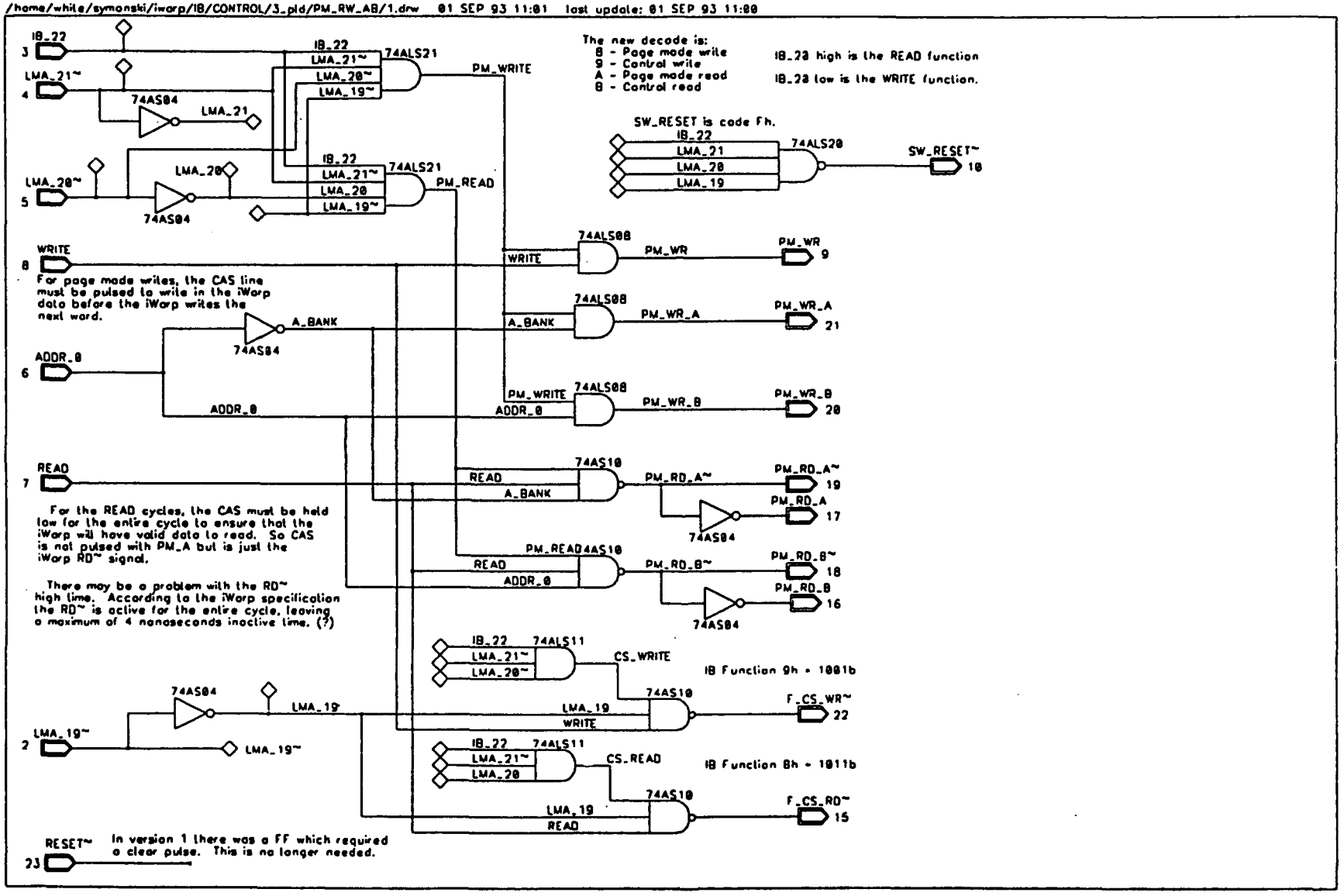


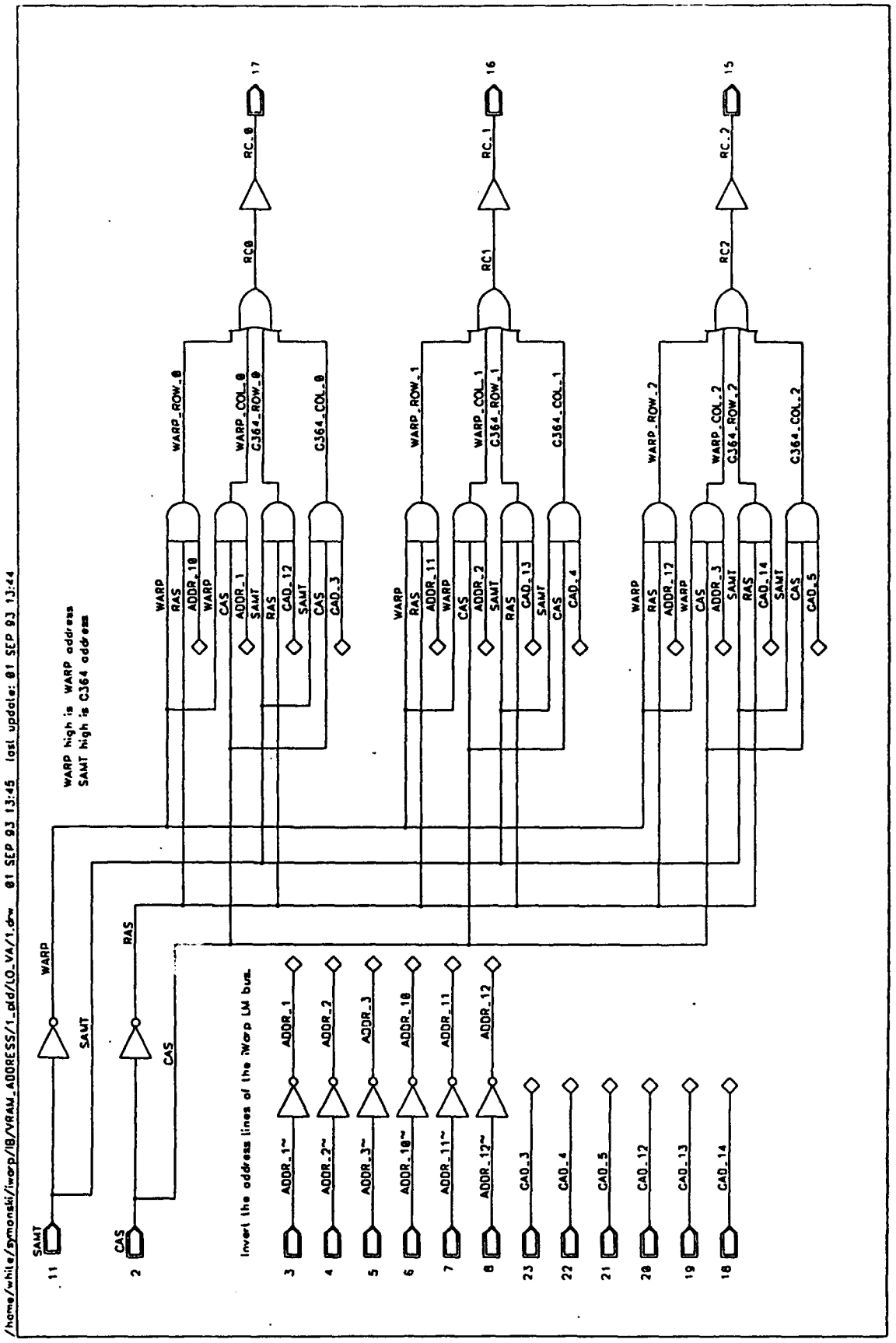
These signals are generated here due to limitations of the PLD devices.

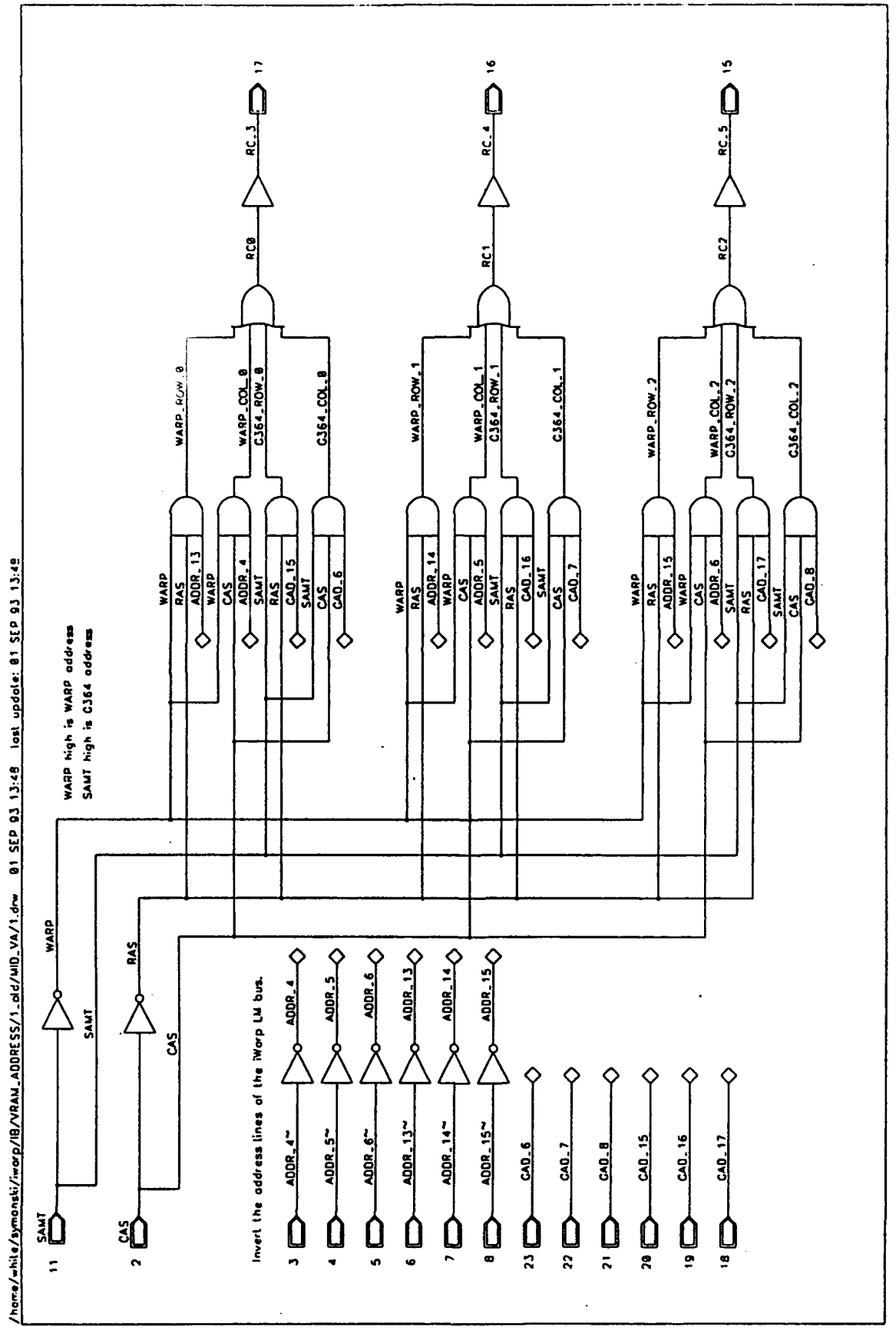
According to the Preliminary specifications, when EVENT goes LOW at the iWarp chip, an interupt is generated. NOTE that the EVENT signal is double inverterted on the OC9 board. So a low here means the iWarp chip sees a low.
Thus, to NOT generate an interupt, select GND as output to IB_EVENT.

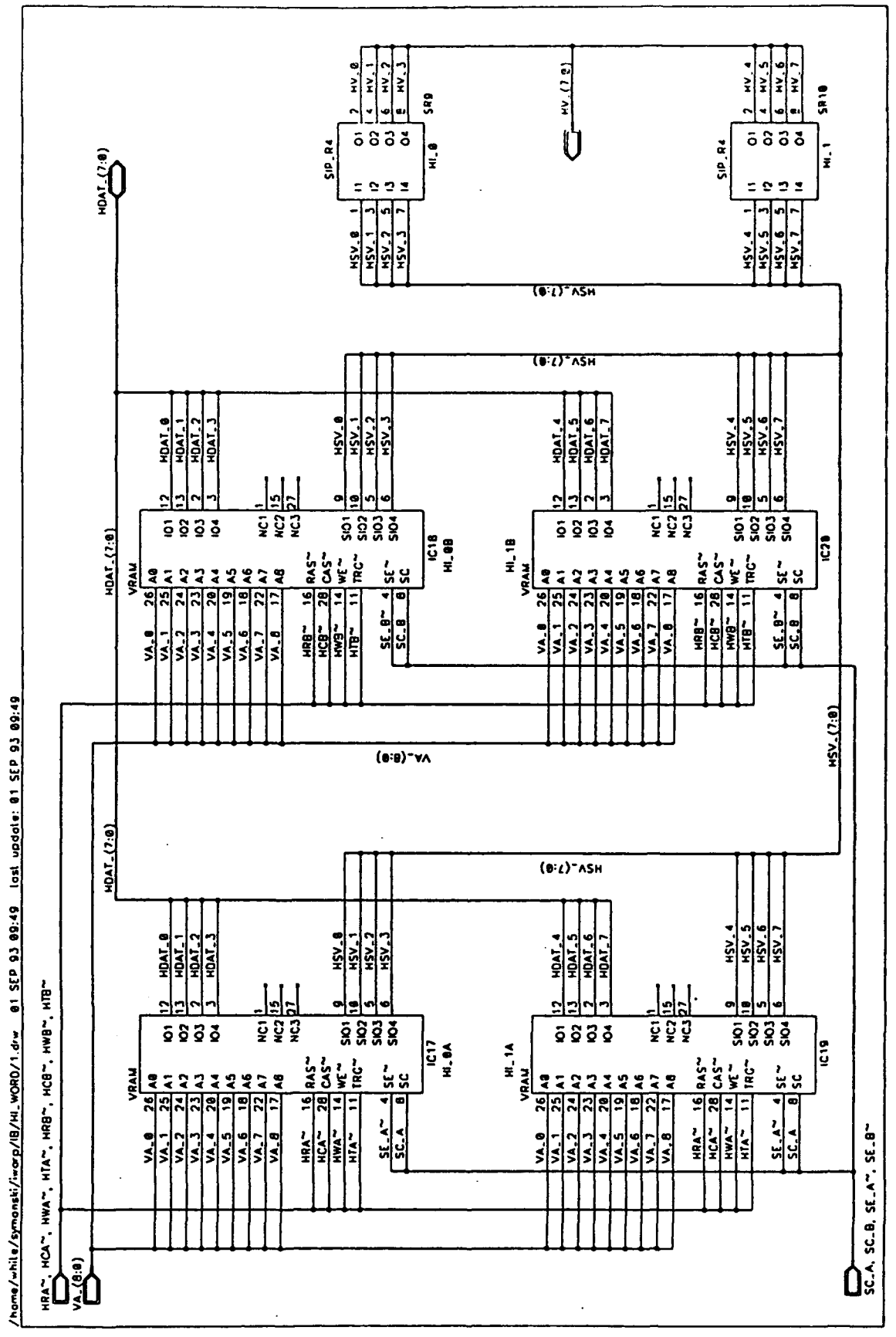
The Serial Access Memory Transfer(SAMT) signal is a better signal to generate an event. Frame Inactive occurs only once each frame and lasts for 500 microseconds. SAMT occrrs every line and data can be sent to the VRAM in page mode for all but 2 microseconds of the 16 microsecond line time. This allows 97% access to the VRAM.

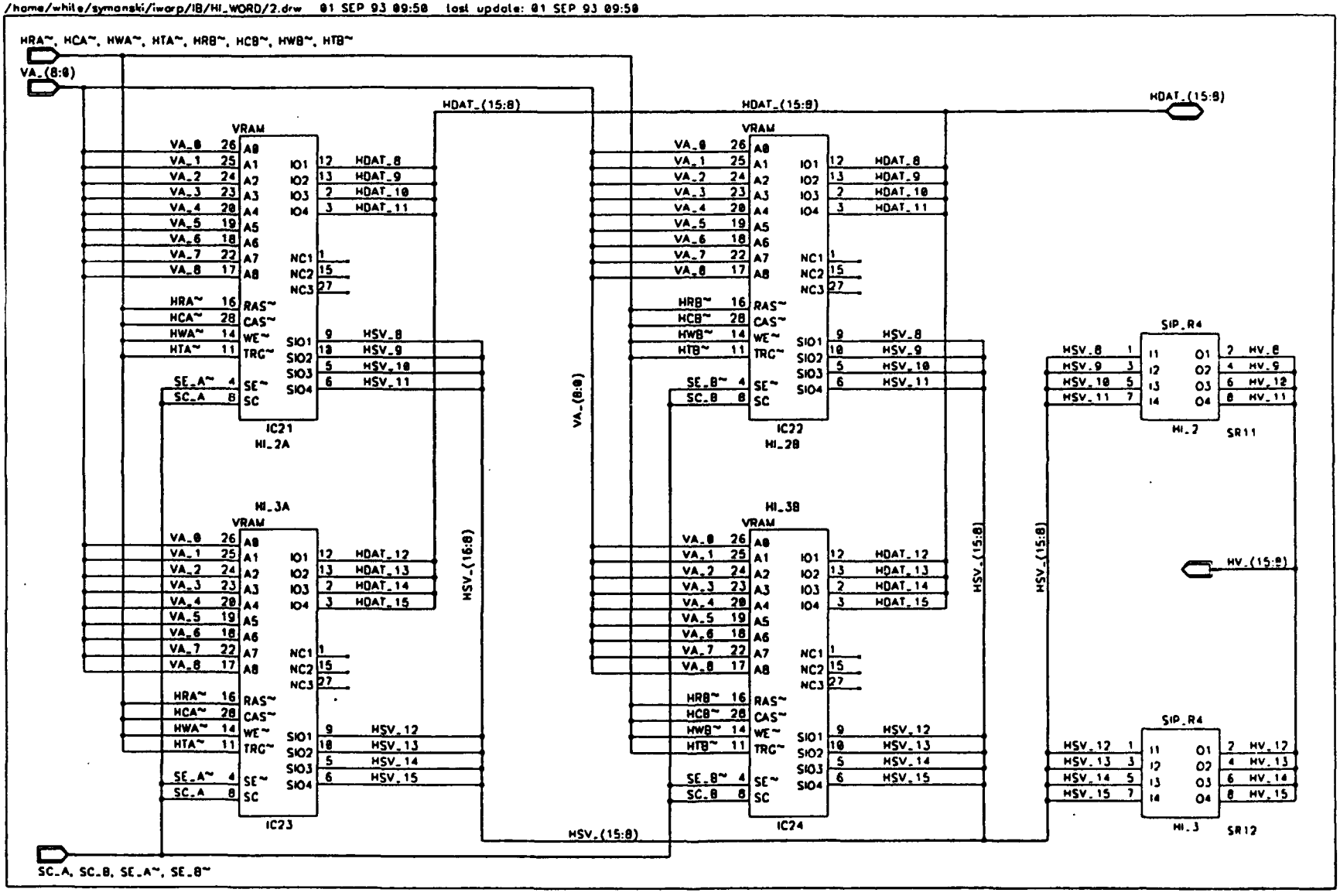
C_RST high resets the C364. Resetting or disabling the C364 upon system reset is a good idea, but since the iWarp performs a system reset when finishing a program, it is better to not disable the C364 and let it run. Thus C_RST[16] will be low at system reset and must be programed high to reset the C364.

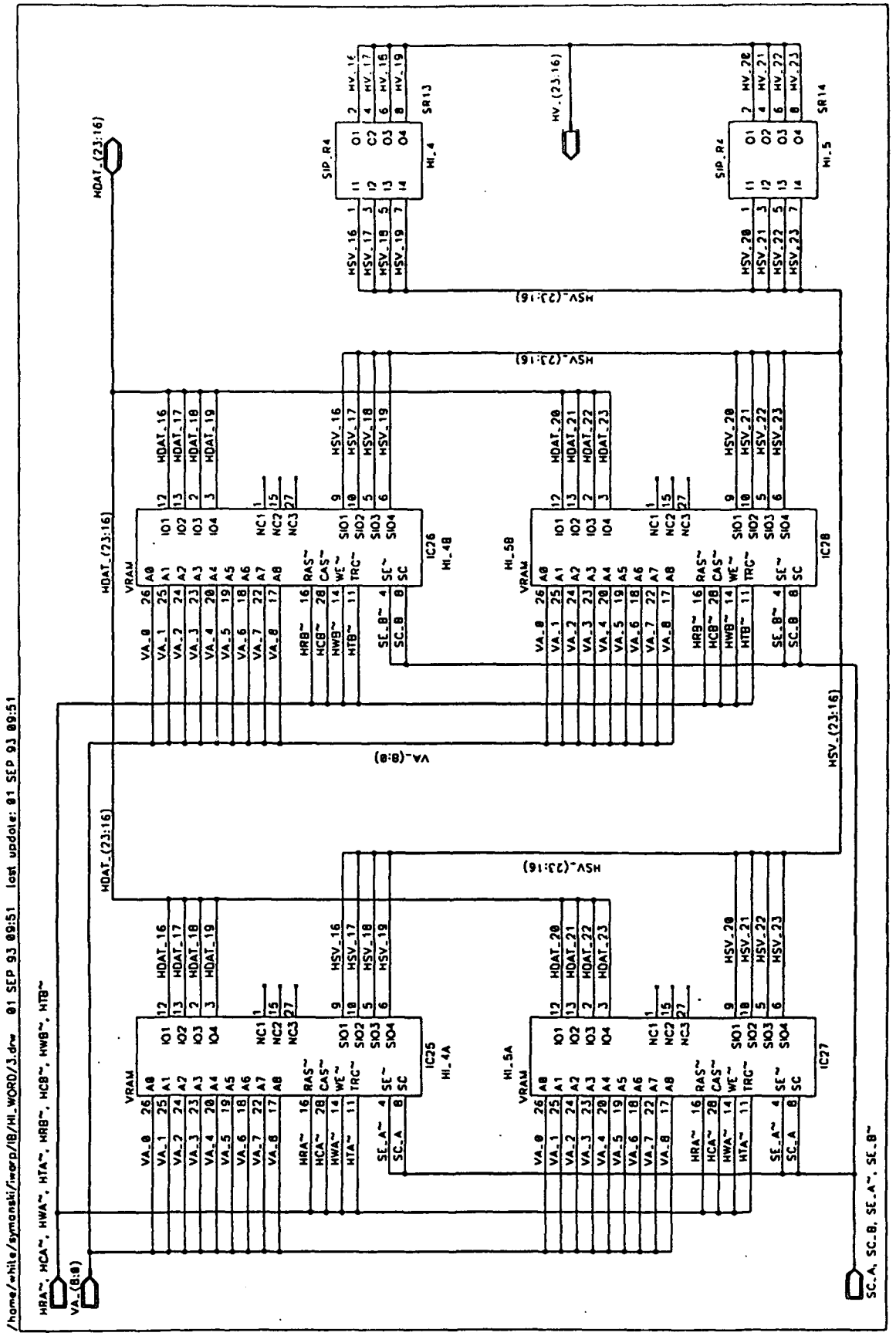

The new decode is:
8 - Page mode write
9 - Control write
A - Page mode read
B - Control read
IB_22 high is the READ function
IB_22 low is the WRITE function.
SW_RESET is code Fh.
IB_22
LMA_21
LMA_20
LMA_19
74ALS20
SW_RESET~
18
IB_22
LMA_21~
LMA_20~
LMA_19~
74ALS21
PM_WRITE
IB_22
LMA_21~
LMA_20
LMA_19~
74ALS21
PM_READ
LMA_21~
4
74AS04
LMA_21
LMA_20~
5
LMA_20
74AS04
WRITE
8
74ALS08
PM_WR
PM_WR
9
For page mode writes, the CAS line must be pulsed to write in the iWarp data before the iWarp writes the next word.
A_BANK
74AS04
74ALS08
PM_WR_A
PM_WR_A
21
ADDR_0
6
ADDR_0
PM_WRITE
74ALS08
PM_WR_B
PM_WR_B
20
READ
7
READ
A_BANK
74AS10
PM_RD_A~
PM_RD_A~
19
PM_RD_A
17
74AS04
For the READ cycles, the CAS must be held low for the entire cycle to ensure that the iWarp will have valid data to read. So CAS is not pulsed with PM_A but is just the iWarp RD~ signal.
There may be a problem with the RD~ high time. According to the iWarp specification the RD~ is active for the entire cycle, leaving a maximum of 4 nanoseconds inactive time. (?)
PM_READ
READ
ADDR_0
74AS10
PM_RD_B~
PM_RD_B~
18
PM_RD_B
16
74AS04
IB_22
LMA_21~
LMA_20~
74ALS11
CS_WRITE
IB Function 9h = 1001b
74AS04
LMA_19
LMA_19
WRITE
74AS10
F_CS_WR~
22
LMA_19~
2
LMA_19~
IB_22
LMA_21~
LMA_20
74ALS11
CS_READ
IB Function Bh = 1011b
LMA_19
READ
74AS10
F_CS_RD~
15
RESET~
23
In version 1 there was a FF which required a clear pulse. This is no longer needed.

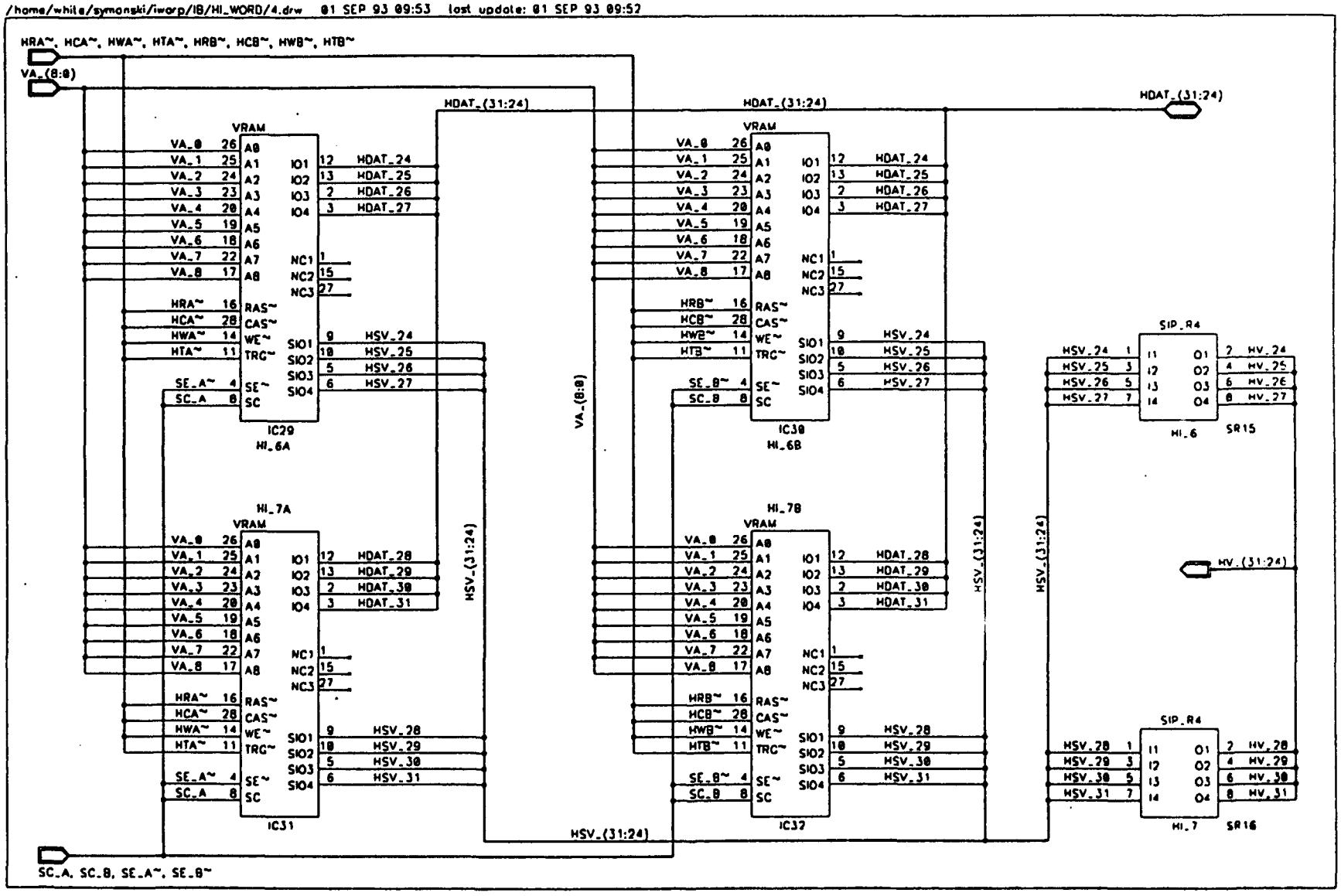

/home/white/symonski/iwarp/IB/VRAM_ADDRESS/1_old/LO_VA/1.drw 01 SEP 93 13:45 last update: 01 SEP 93 13:44
WARP high is WARP address
SAMT high is C364 address
Invert the address lines of the iWarp LM bus.
SAMT
CAS
WARP
SAMT
RAS
CAS
ADDR_1~
ADDR_2~
ADDR_3~
ADDR_10~
ADDR_11~
ADDR_12~
CAD_3
CAD_4
CAD_5
CAD_12
CAD_13
CAD_14
RC0
RC1
RC2
RC_0
RC_1
RC_2
WARP_ROW_0
WARP_COL_0
C364_ROW_0
C364_COL_0
WARP_ROW_1
WARP_COL_1
C364_ROW_1
C364_COL_1
WARP_ROW_2
WARP_COL_2
C364_ROW_2
C364_COL_2
WARP
RAS
ADDR_10
ADDR_1
CAS
SAMT
RAS
CAD_12
SAMT
CAS
CAD_3
WARP
RAS
ADDR_11
ADDR_2
CAS
SAMT
RAS
CAD_13
SAMT
CAS
CAD_4
WARP
RAS
ADDR_12
ADDR_3
CAS
SAMT
RAS
CAD_14
SAMT
CAS
CAD_5
11
2
3
4
5
6
7
8
23
22
21
20
19
18
17
16
15

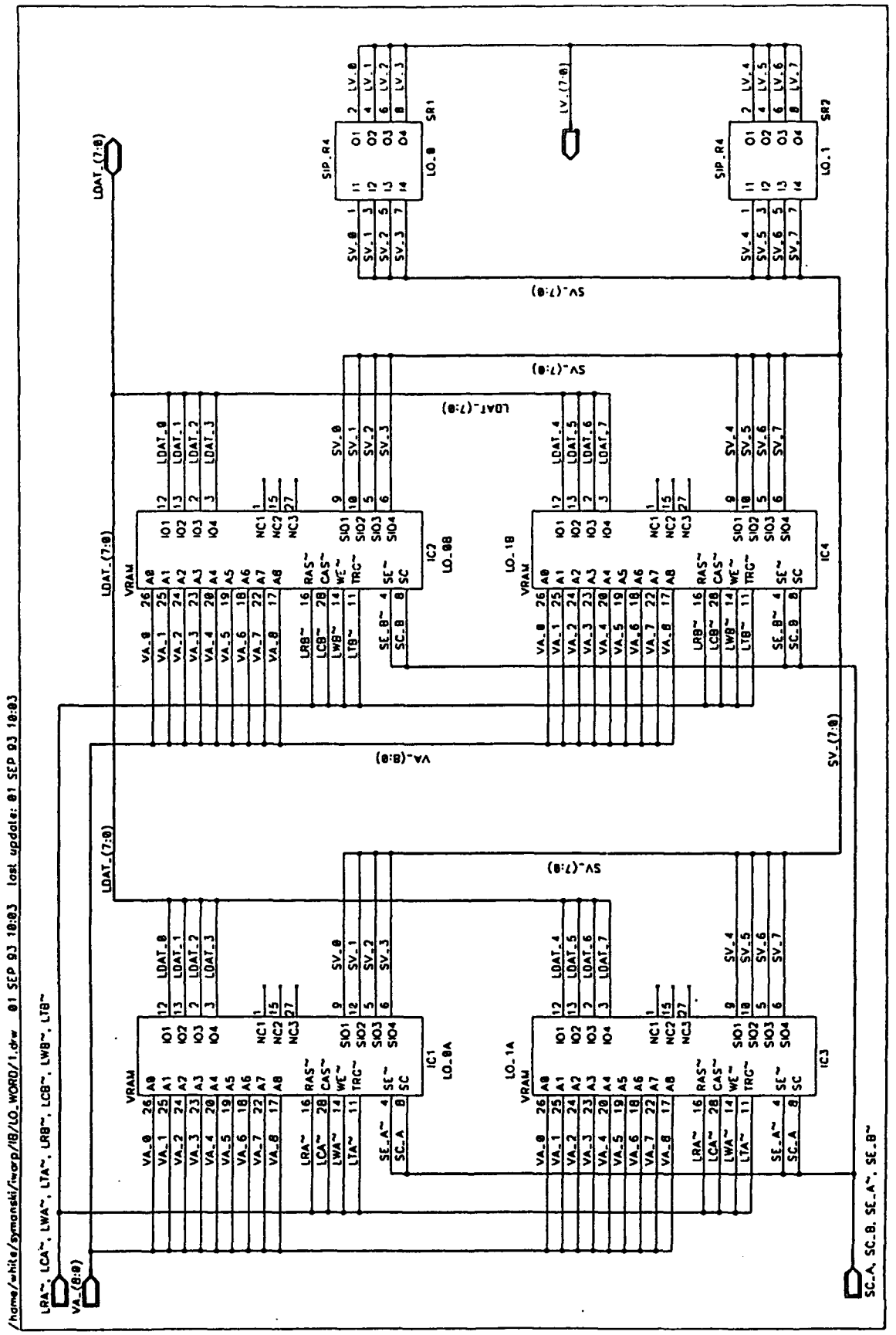

/home/white/symonski/iwarp/IB/VRAM_ADDRESS/1_old/WID_VA/1.drw 01 SEP 93 13:48 last update: 01 SEP 93 13:48
WARP high is WARP address
SAMT high is C364 address
Invert the address lines of the iWarp LM bus.
WARP
SAMT
RAS
CAS
WARP_ROW_0
WARP_COL_0
C364_ROW_0
C364_COL_0
WARP_ROW_1
WARP_COL_1
C364_ROW_1
C364_COL_1
WARP_ROW_2
WARP_COL_2
C364_ROW_2
C364_COL_2
RC0
RC1
RC2
RC_3
RC_4
RC_5
ADDR_4
ADDR_5
ADDR_6
ADDR_13
ADDR_14
ADDR_15
ADDR_4~
ADDR_5~
ADDR_6~
ADDR_13~
ADDR_14~
ADDR_15~
CAD_6
CAD_7
CAD_8
CAD_15
CAD_16
CAD_17

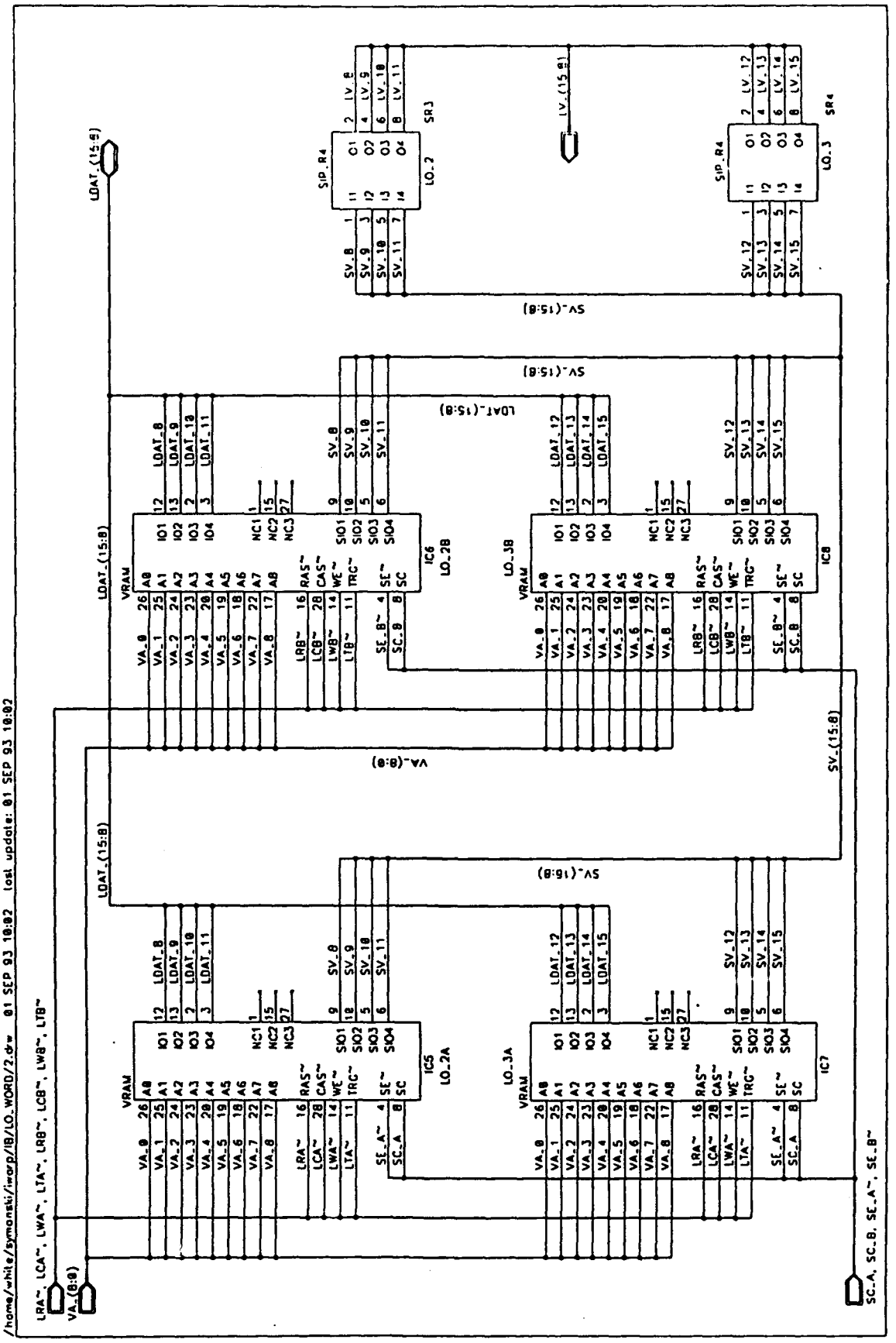

/home/white/symonski/iwarp/IB/HI_WORD/1.drw 01 SEP 93 09:49 last update: 01 SEP 93 09:49
HRA~, HCA~, HWA~, HTA~, HRB~, HCB~, HWB~, HTB~
VA_(8:0)
HDAT_(7:0)
HDAT_(7:0)
HDAT_(7:0)
VA_(8:0)
HSV_(7:0)
HSV_(7:0)
HSV_(7:0)
HSV_(7:0)
HV_(7:0)
SC_A, SC_B, SE_A~, SE_B~
SIP_R4
SR9
SR10
HI_0
HI_1
VRAM
IC17
IC18
IC19
IC20
HI_0A
HI_0B
HI_1A
HI_1B

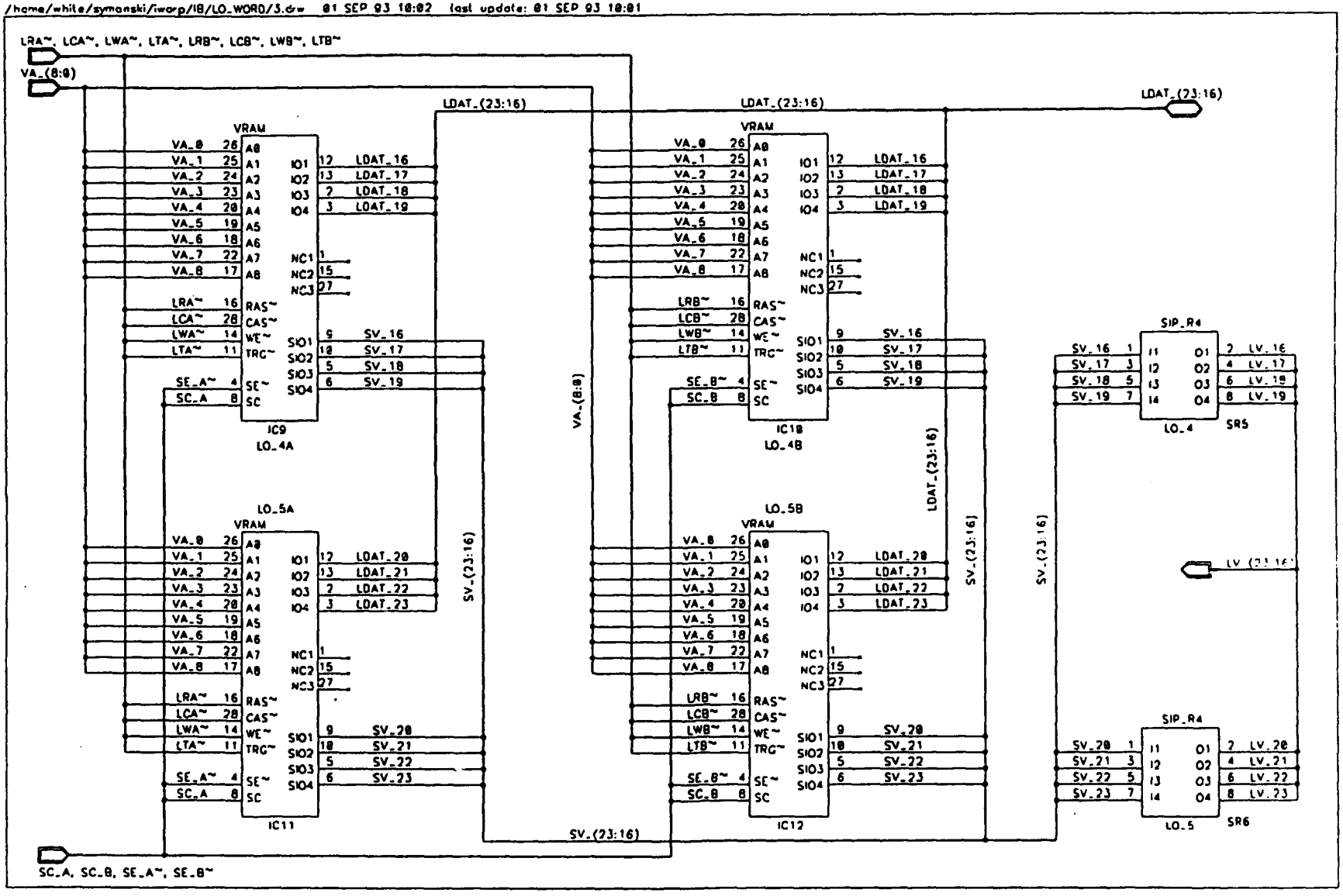

/home/white/symanski/iwarp/IB/HI_WORD/2.drw 01 SEP 93 09:58 last update: 01 SEP 93 09:58
HRA~, HCA~, HWA~, HTA~, HRB~, HCB~, HWB~, HTB~
VA_(8:0)
HDAT_(15:8)
HDAT_(15:8)
HDAT_(15:8)
VRAM
IC21
HI_2A
VRAM
IC22
HI_2B
SIP_R4
HI_2
SR11
HI_3A
VRAM
IC23
HI_3B
VRAM
IC24
SIP_R4
HI_3
SR12
HV_(15:8)
HSV_(16:8)
HSV_(15:8)
HSV_(15:8)
HSV_(15:8)
VA_(8:0)
SC_A, SC_B, SE_A~, SE_B~

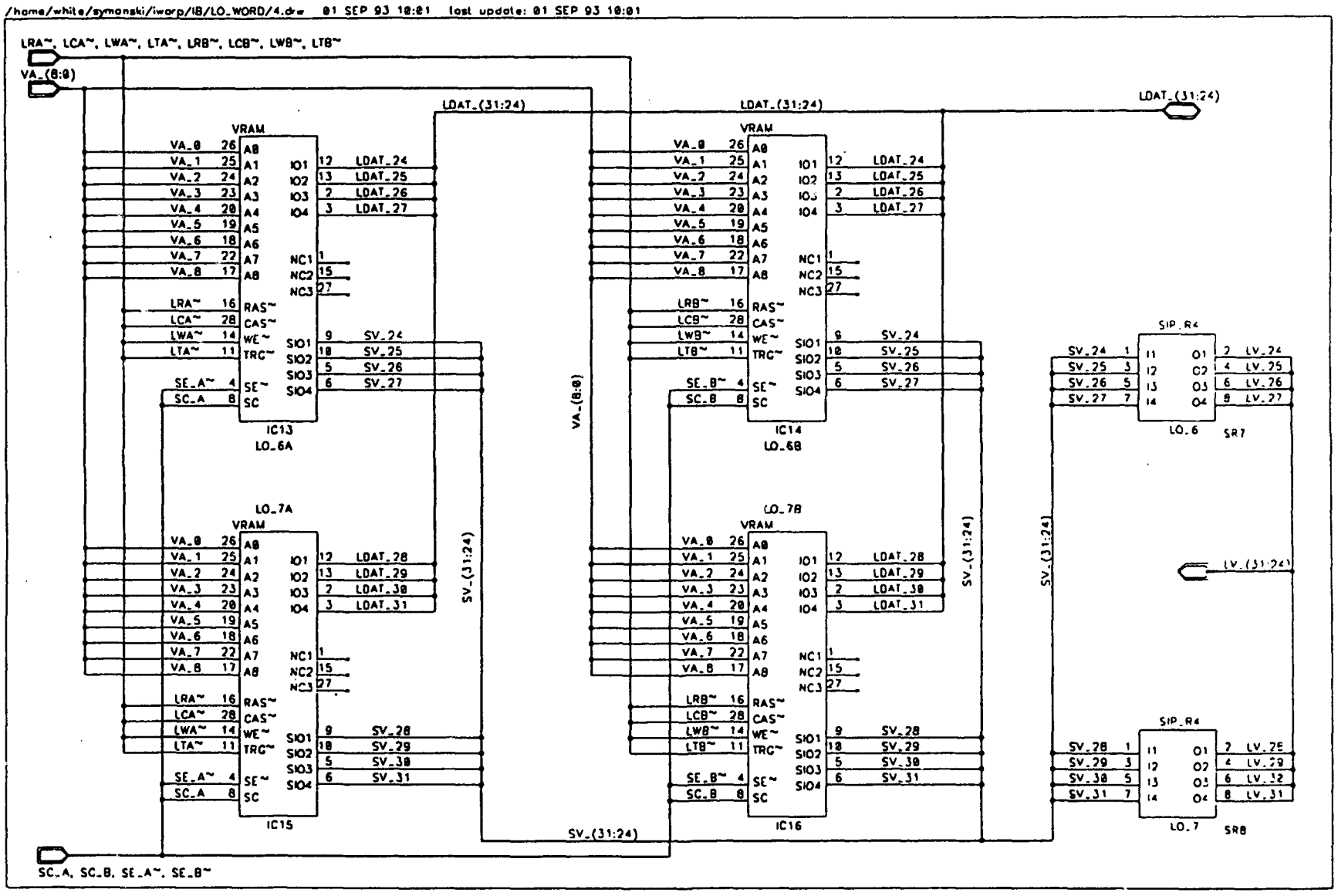

/home/white/symanski/imerp/18/HI_WORD/3.drw 01 SEP 93 09:51 last update: 01 SEP 93 09:51
HRA~, HCA~, HWA~, HTA~, HRB~, HCB~, HWB~, HTB~
VA_(8:0)
SC_A, SC_B, SE_A~, SE_B~
HDAT_(23:16)
HSV_(23:16)
VA_(8:0)
HV_(23:16)
VRAM
IC25
HI_4A
IC26
HI_4B
HI_5A
IC27
HI_5B
IC28
SIP_R4
SR13
HI_4
SIP_R4
SR14
HI_5

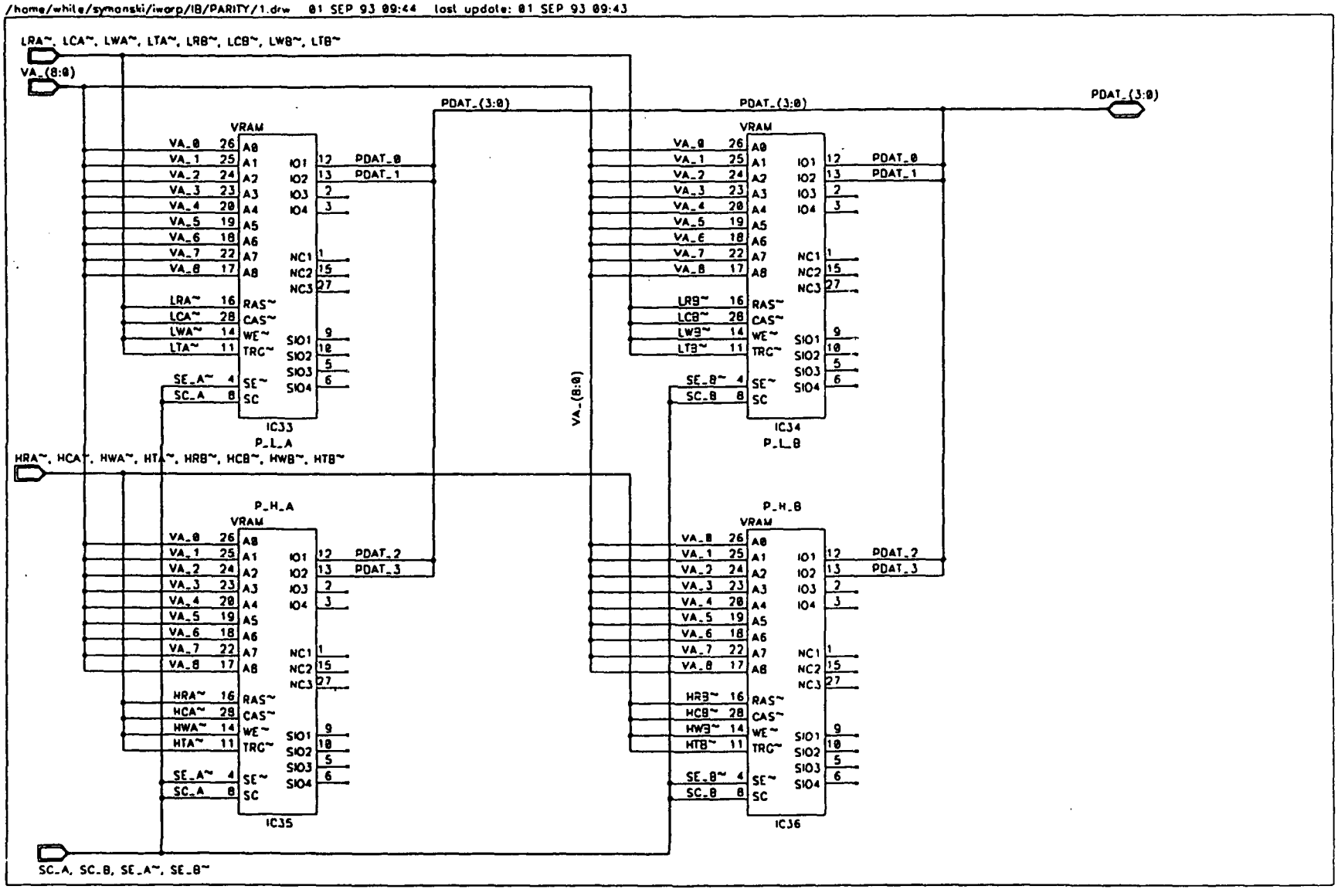

/home/white/symonski/iworp/IB/HI_WORD/4.drw 01 SEP 93 09:53 last update: 01 SEP 93 09:52
HRA~, HCA~, HWA~, HTA~, HRB~, HCB~, HWB~, HTB~
VA_(8:0)
HDAT_(31:24)
VRAM
IC29
HI_6A
IC30
HI_6B
HI_7A
IC31
HI_7B
IC32
SIP_R4
HI_6
SR15
HI_7
SR16
HV_(31:24)
HSV_(31:24)
SC_A, SC_B, SE_A~, SE_B~

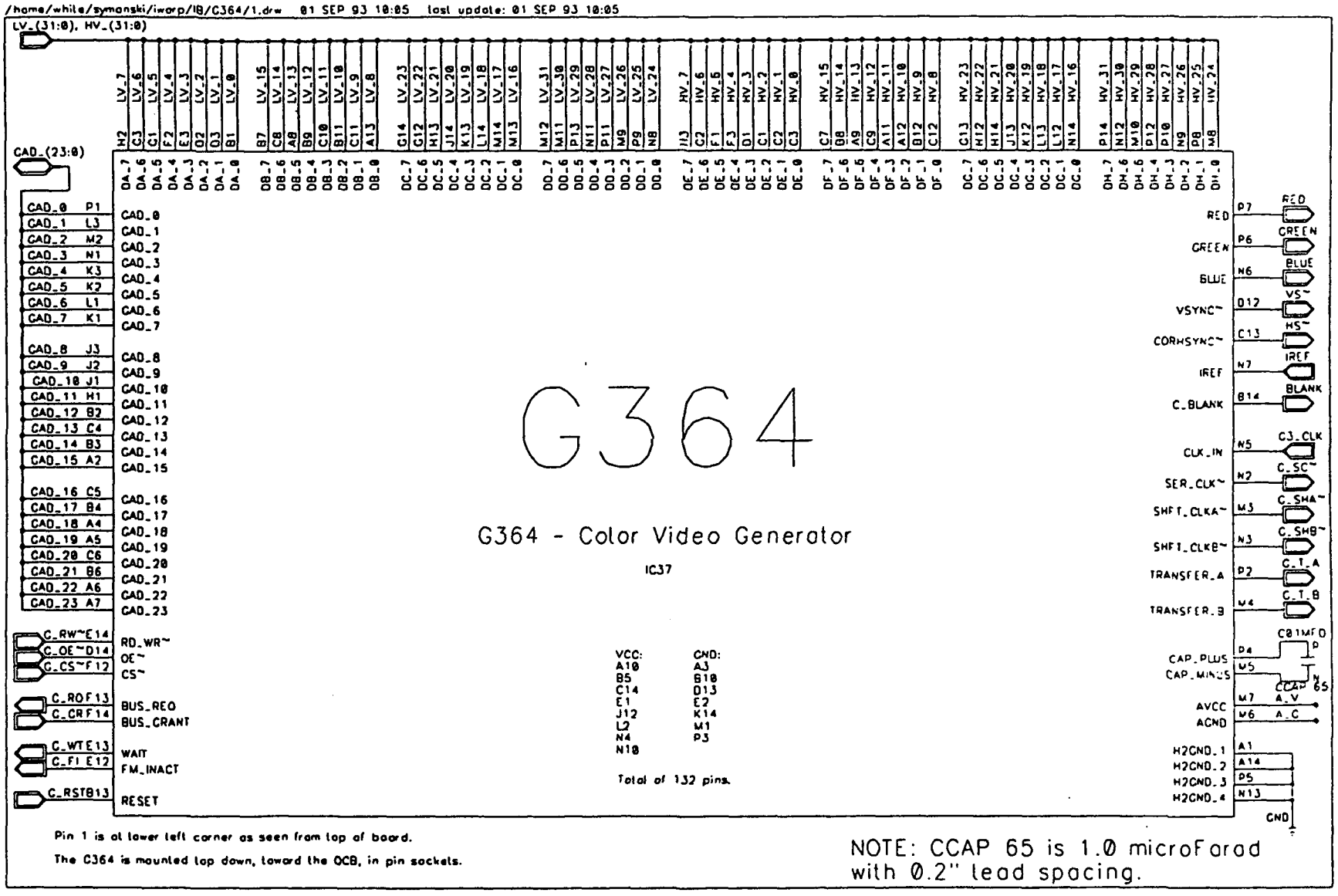

/home/white/symonski/iworp/1B/LO_WORD/1.drw 01 SEP 93 10:03 last update: 01 SEP 93 10:03
LRA~, LCA~, LWA~, LTA~, LRB~, LCB~, LWB~, LTB~
VA_(8:0)
LDAT_(7:0)
SC_A, SC_B, SE_A~, SE_B~
SV_(7:0)
LV_(7:0)
VRAM
SIP_R4
IC1
IC2
IC3
IC4
SR1
SR2


/home/white/symonski/iworp/IB/LO_WORD/2.drw 01 SEP 93 10:02 last update: 01 SEP 93 10:02
LRA~, LCA~, LWA~, LTA~, LRB~, LCB~, LWB~, LTB~
VA_(8:0)
LOAT_(15:8)
SV_(15:8)
LV_(15:8)
SC_A, SC_B, SE_A~, SE_B~
LO_2B
LO_3B
LO_2A
LO_3A
IC6
IC8
IC5
IC7
SR3
SR4
SIP_R4
VRAM

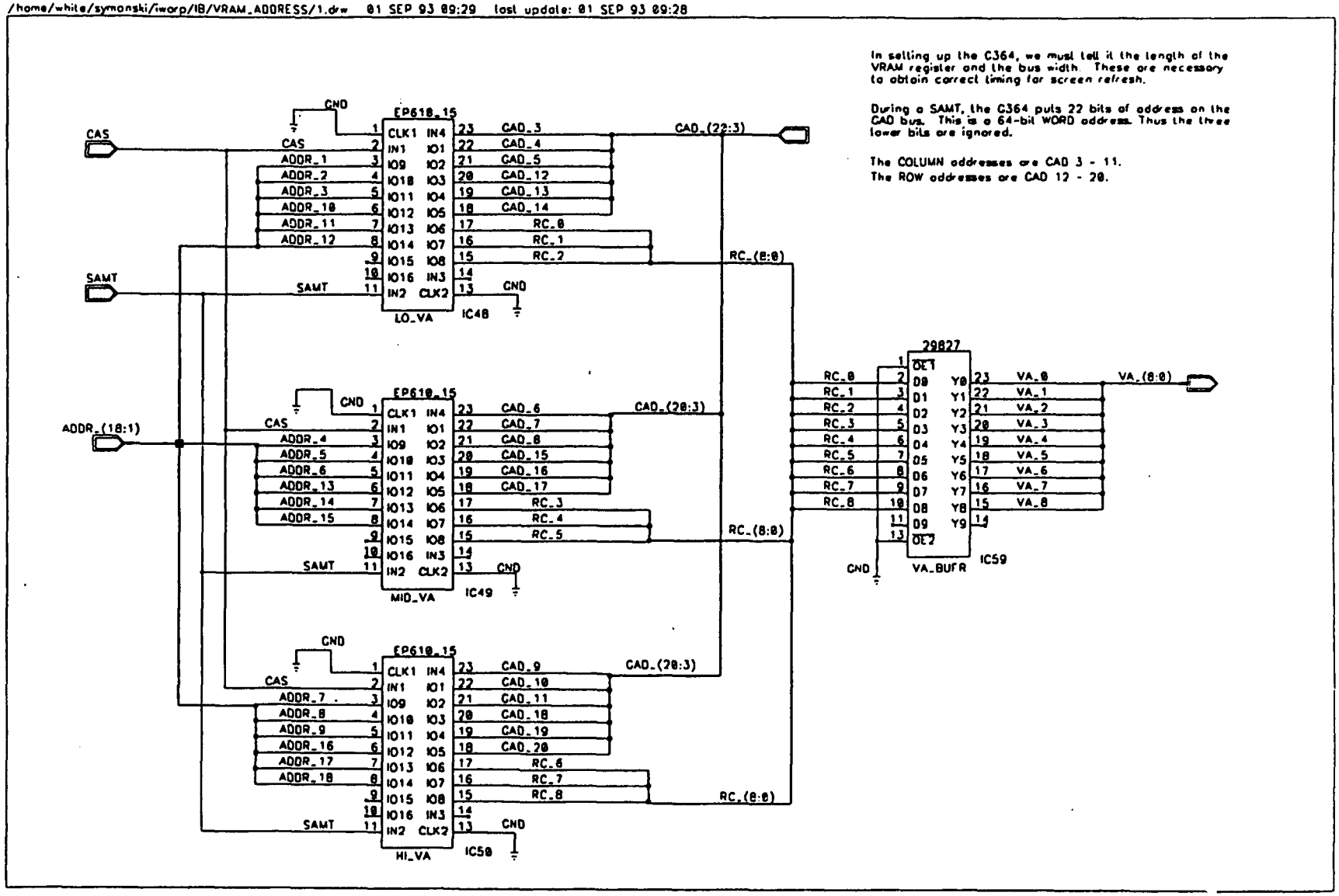

/home/white/symanski/iwarp/18/LO_WORD/3.drw 01 SEP 93 10:02 last update: 01 SEP 93 10:01
LRA~, LCA~, LWA~, LTA~, LRB~, LCB~, LWB~, LTB~
VA_(8:0)
LDAT_(23:16)
LDAT_(23:16)
LDAT_(23:16)
VRAM
IC9
LO_4A
VRAM
IC10
LO_4B
LO_5A
VRAM
IC11
LO_5B
VRAM
IC12
SIP_R4
LO_4
SR5
SIP_R4
LO_5
SR6
VA_(8:0)
SV_(23:16)
LDAT_(23:16)
SV_(23:16)
SV_(23:16)
SV_(23:16)
LV_(23:16)
SC_A, SC_B, SE_A~, SE_B~

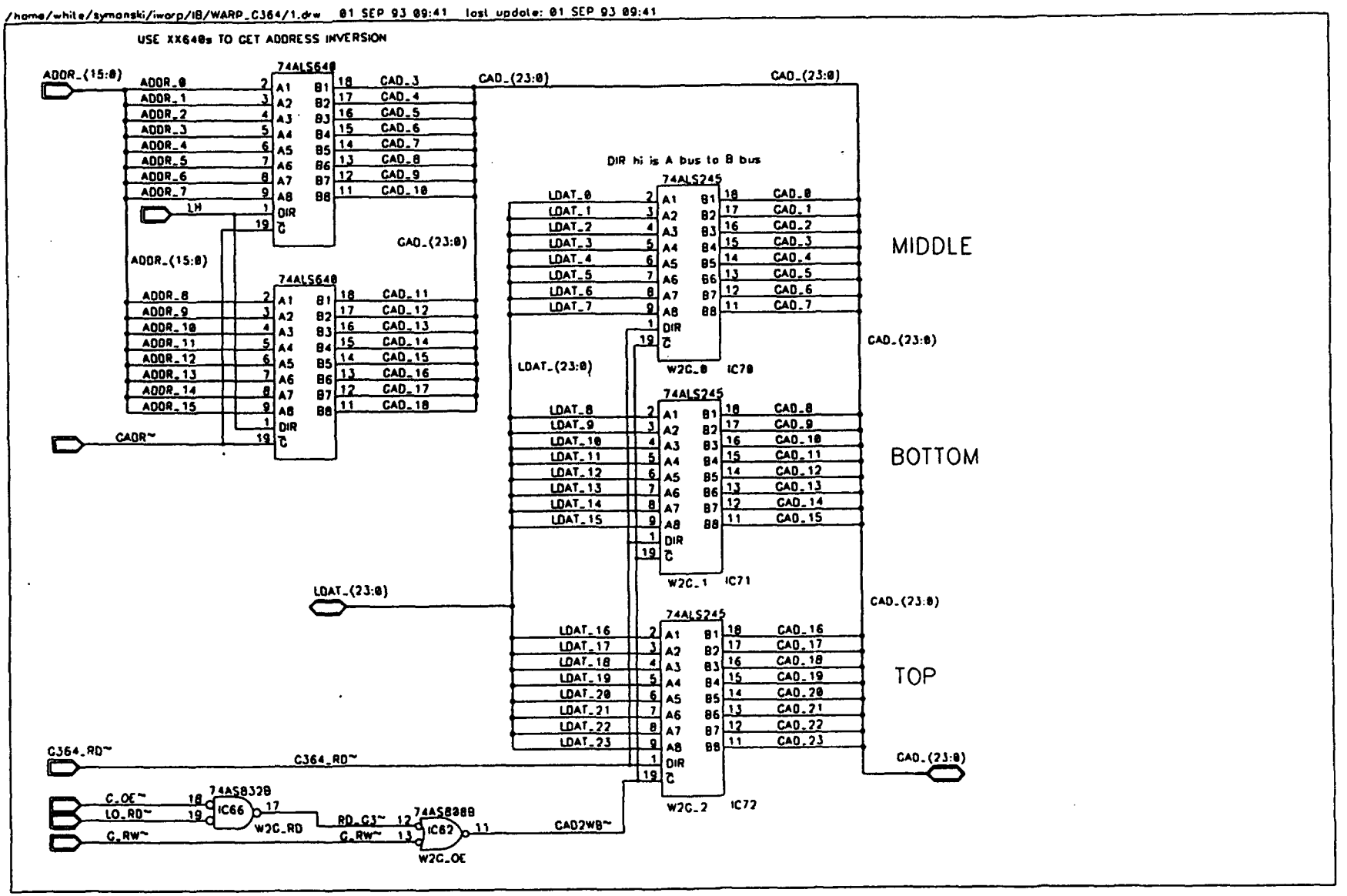

/home/white/symonski/iwarp/IB/LO_WORD/4.dw 01 SEP 93 10:01 last update: 01 SEP 93 10:01
LRA~, LCA~, LWA~, LTA~, LRB~, LCB~, LWB~, LTB~
VA_(8:0)
LDAT_(31:24)
VRAM
IC13
LO_6A
IC14
LO_6B
LO_7A
IC15
LO_7B
IC16
SIP_R4
LO_6
SR7
LO_7
SR8
SV_(31:24)
LV_(31:24)
SC_A, SC_B, SE_A~, SE_B~

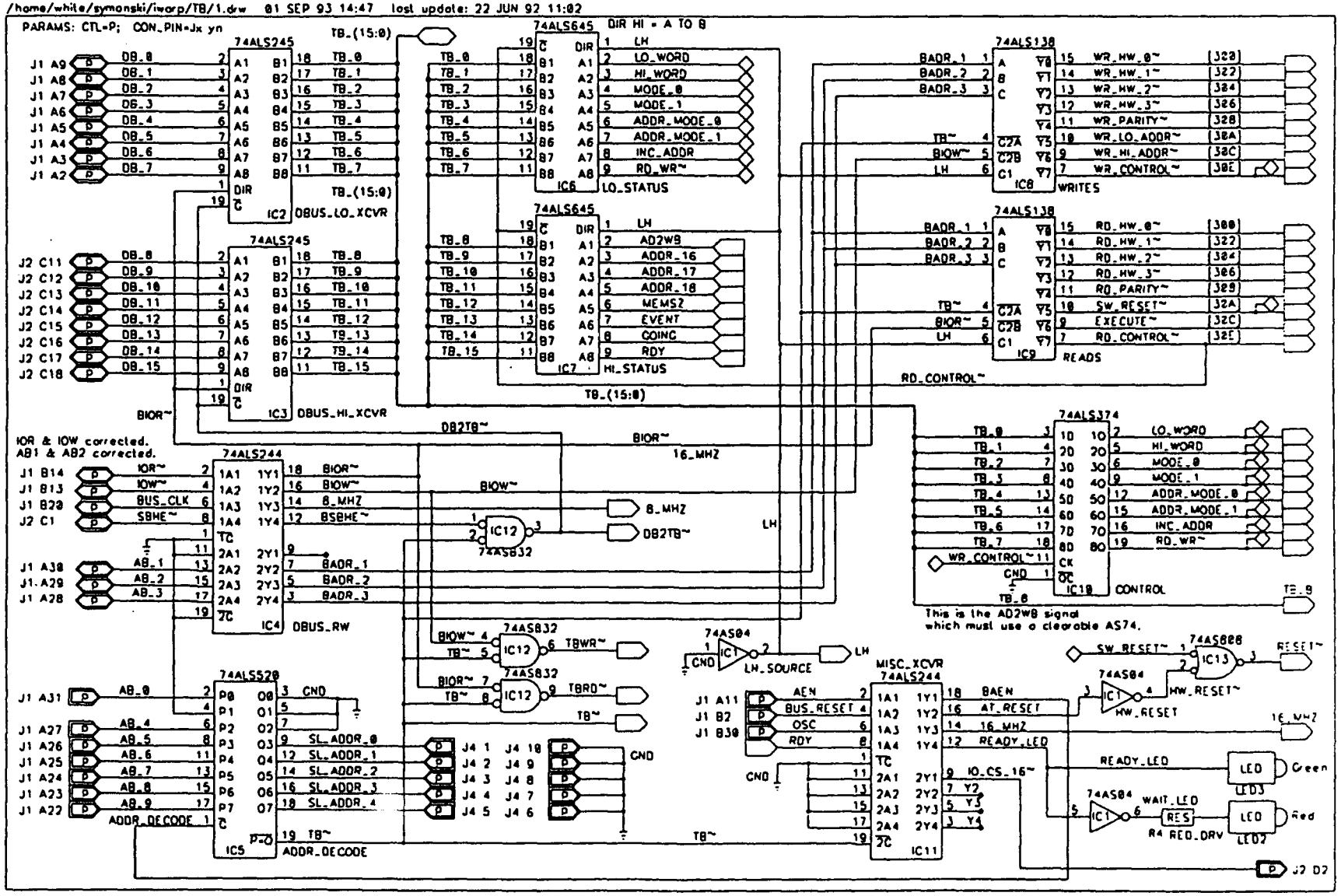

/home/white/symonski/iworp/IB/PARITY/1.drw 01 SEP 93 09:44 last update: 01 SEP 93 09:43
LRA~, LCA~, LWA~, LTA~, LRB~, LCB~, LWB~, LTB~
VA_(8:0)
PDAT_(3:0)
PDAT_(3:0)
PDAT_(3:0)
VRAM
IC33
P_L_A
VRAM
IC34
P_L_B
HRA~, HCA~, HWA~, HTA~, HRB~, HCB~, HWB~, HTB~
P_H_A
VRAM
IC35
P_H_B
VRAM
IC36
SC_A, SC_B, SE_A~, SE_B~

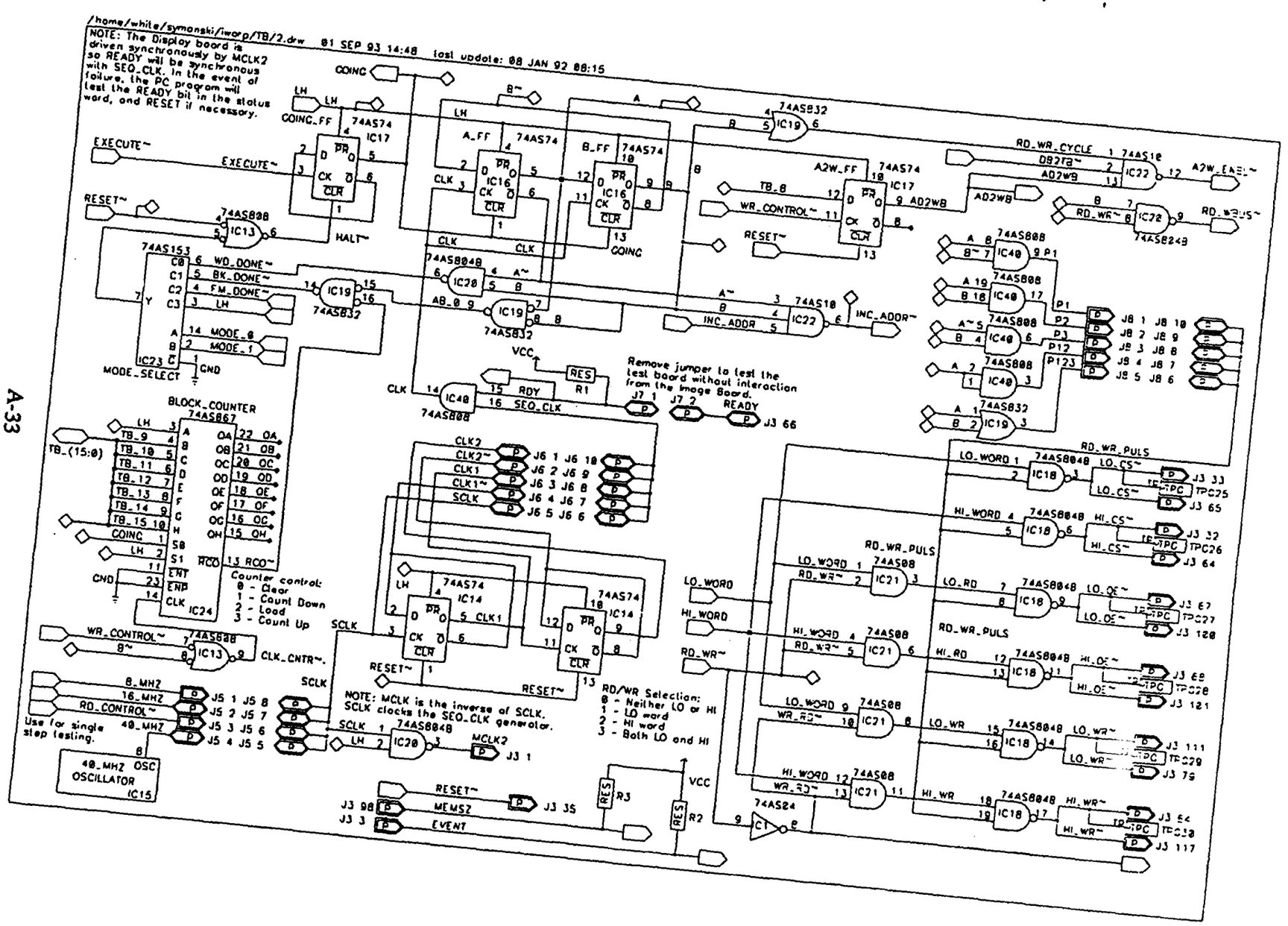

/home/white/symanski/iworp/IB/C364/1.drw 01 SEP 93 10:05 last update: 01 SEP 93 10:05
LV_(31:0), HV_(31:0)
CAD_(23:0)
G364
G364 - Color Video Generator
IC37
VCC:
A10
B5
C14
E1
J12
L2
N4
N10
GND:
A3
B10
D13
E2
K14
M1
P3
Total of 132 pins.
RED
GREEN
BLUE
VSYNC~
CORHSYNC~
IREF
C_BLANK
CLK_IN
SER_CLK~
SHFT_CLKA~
SHFT_CLKB~
TRANSFER_A
TRANSFER_B
CAP_PLUS
CAP_MINUS
AVCC
ACND
H2GND_1
H2GND_2
H2GND_3
H2GND_4
GND
C81MFD
CCAP 65
RD_WR~
OE~
CS~
BUS_REQ
BUS_GRANT
WAIT
FM_INACT
RESET
Pin 1 is at lower left corner as seen from top of board.
The C364 is mounted top down, toward the OCB, in pin sockets.
NOTE: CCAP 65 is 1.0 microFarad with 0.2" lead spacing.

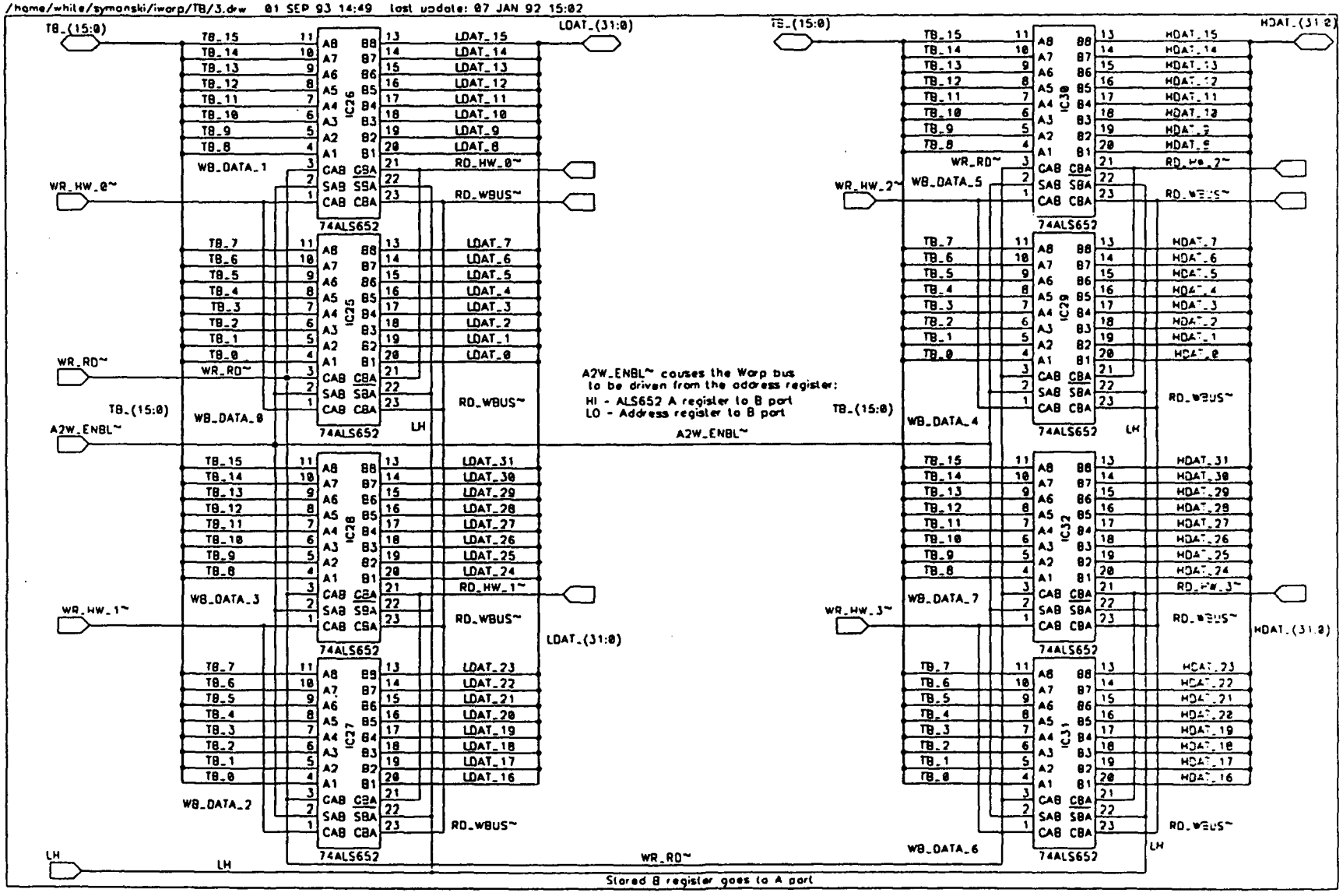
/home/white/symonski/iworp/IB/VRAM_ADDRESS/1.drw 01 SEP 93 09:29 last update: 01 SEP 93 09:28

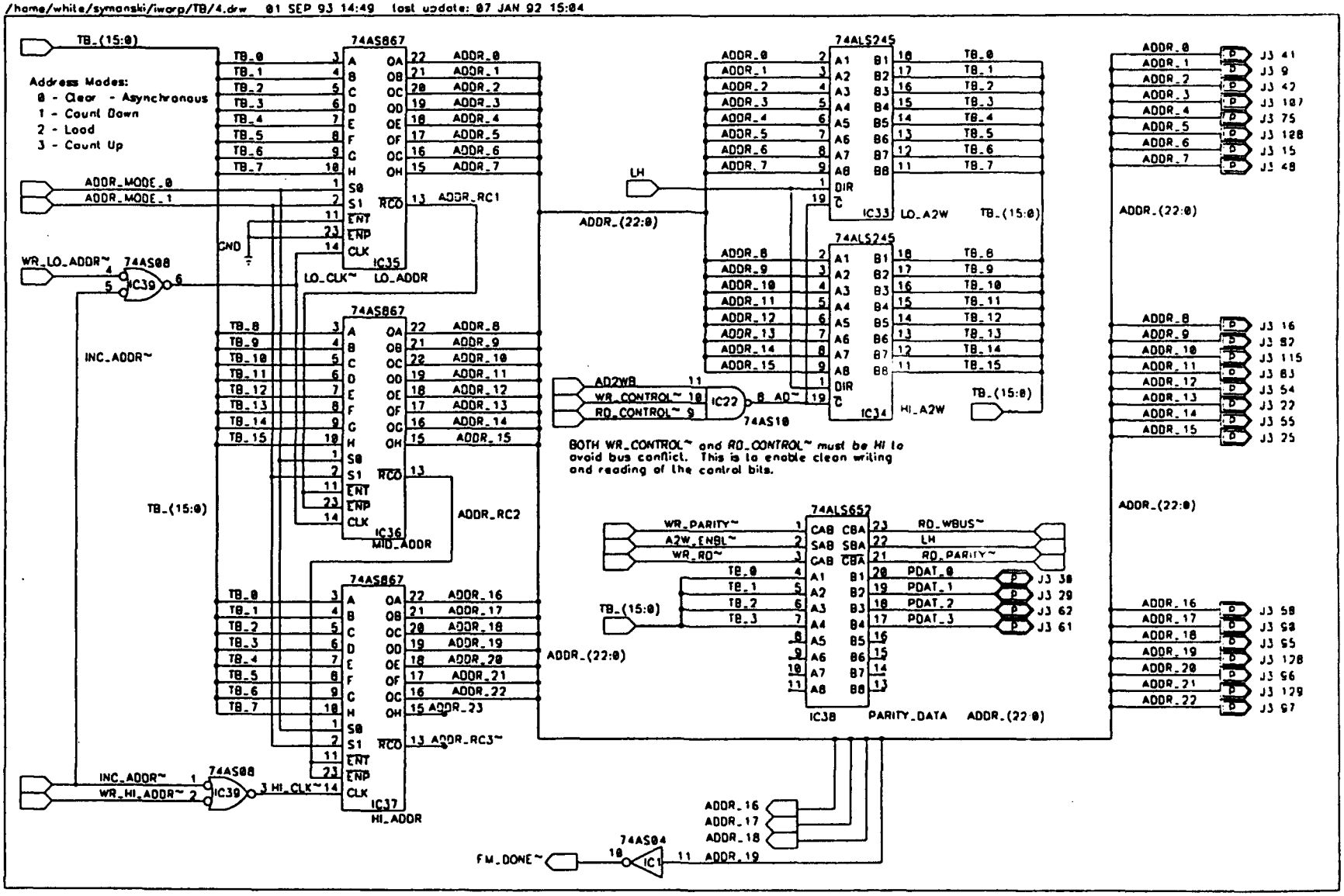


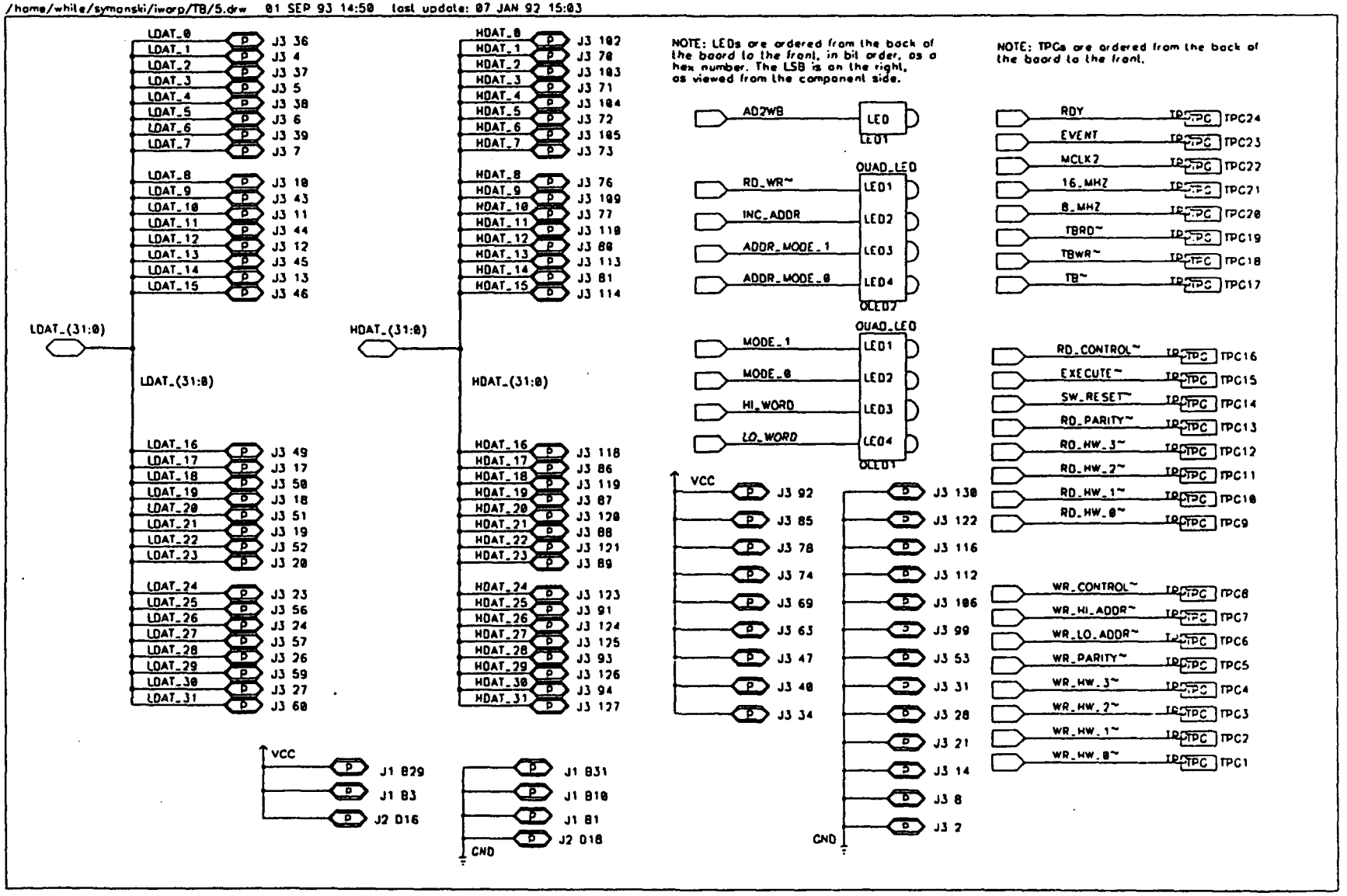

/home/white/symanski/iwarp/IB/WARP_C364/1.drw 01 SEP 93 09:41 last update: 01 SEP 93 09:41
USE XX640s TO GET ADDRESS INVERSION
ADDR_(15:0)
74ALS640
CAD_(23:0)
DIR hi is A bus to B bus
74ALS245
MIDDLE
BOTTOM
TOP
LDAT_(23:0)
W2C_0 IC70
W2C_1 IC71
W2C_2 IC72
LH
CADR~
C364_RD~
74AS8328
74AS8288
IC66
IC62
W2C_RD
W2C_OE
C_OE~
LO_RD~
C_RW~
RD_C3~
CAD2WB~

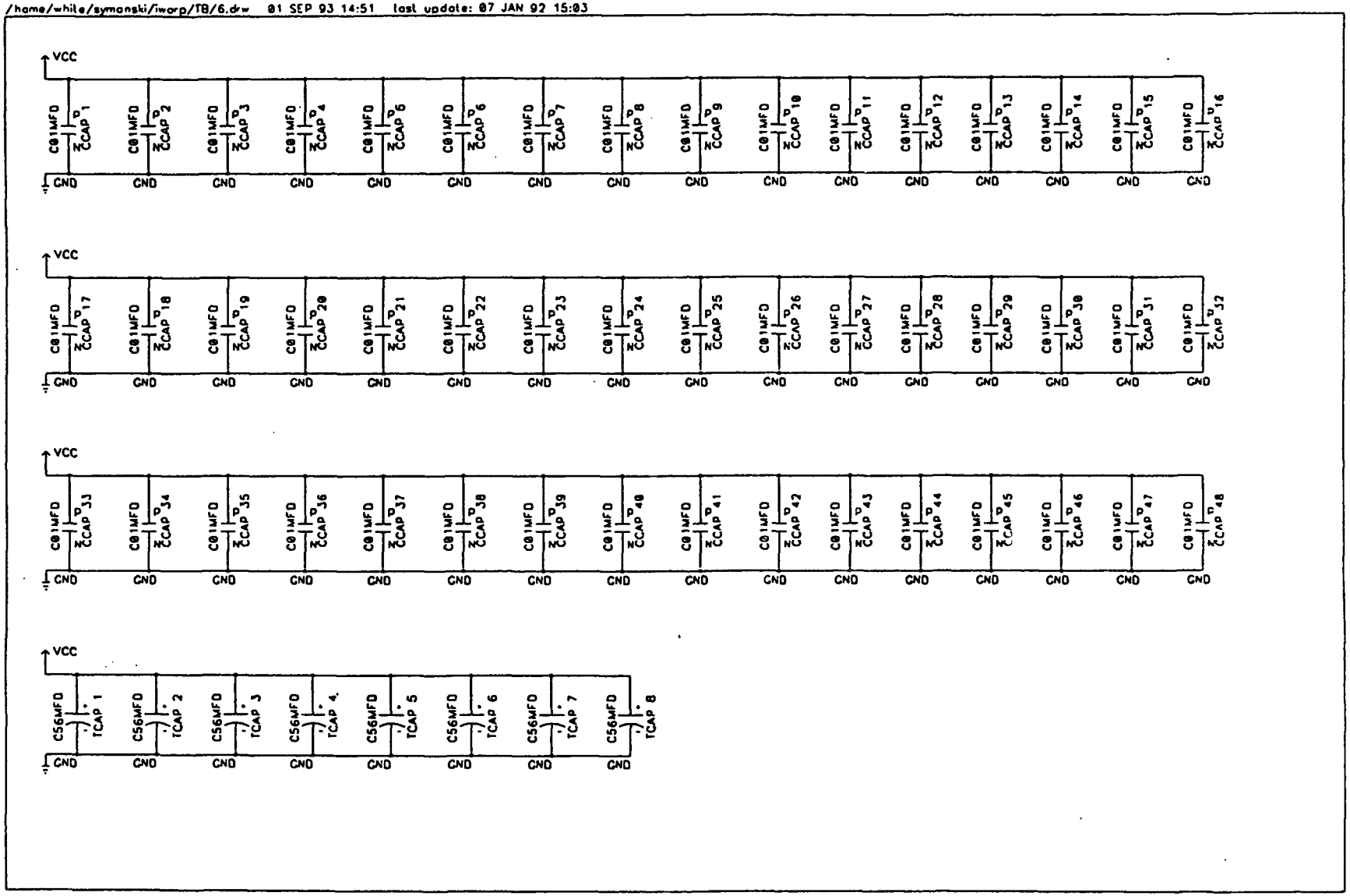

/home/white/symonski/iworp/TB/1.drw 01 SEP 93 14:47 last update: 22 JUN 92 11:02
PARAMS: CTL-P; CON_PIN-Jx yn
74ALS245
74ALS645
DIR HI = A TO B
74ALS138
74ALS374
74ALS244
74ALS520
74AS832
74AS08
74AS04
74AS808
LO_STATUS
HI_STATUS
WRITES
READS
CONTROL
DBUS_LO_XCVR
DBUS_HI_XCVR
DBUS_RW
ADDR_DECODE
LH_SOURCE
MISC_XCVR
IOR & IOW corrected.
AB1 & AB2 corrected.
This is the AD2WB signal which must use a clearable AS74.
16_MHZ
8_MHZ
BIOR~
BIOW~
LH
RD_CONTROL~
WR_CONTROL~
HW_RESET
READY_LED
WAIT_LED
Green
Red
J2 D2


/home/white/symonski/iwarp/TB/2.dw 01 SEP 93 14:48 last update: 08 JAN 92 08:15
NOTE: The Display board is driven synchronously by MCLK2 so READY will be synchronous with SEQ_CLK. In the event of failure, the PC program will test the READY bit in the status word, and RESET if necessary.
Remove jumper to test the test board without interaction from the Image Board.
BLOCK_COUNTER
74AS867
Counter control:
0 - Clear
1 - Count Down
2 - Load
3 - Count Up
RD/WR Selection:
0 - Neither LO or HI
1 - LO word
2 - HI word
3 - Both LO and HI
NOTE: MCLK is the inverse of SCLK. SCLK clocks the SEQ_CLK generator.
Use for single step testing.
40_MHZ OSC OSCILLATOR IC15
74AS153
74AS74
74AS832
74AS808
74AS804B
74AS18
74AS08
74AS24
74AS245
74AS518
RD_WR_PULS
MODE_SELECT
INC_ADDR~
A2W_ENABLE~
RD_WR_BUS~


/home/white/symanski/iwarp/TB/3.drw 01 SEP 93 14:49 last update: 07 JAN 92 15:02
TB_(15:0)
LDAT_(31:0)
HDAT_(31:0)
IC26
IC25
IC28
IC27
IC30
IC29
IC32
IC31
74ALS652
WB_DATA_1
WB_DATA_0
WB_DATA_3
WB_DATA_2
WB_DATA_5
WB_DATA_4
WB_DATA_7
WB_DATA_6
WR_HW_0~
WR_HW_1~
WR_HW_2~
WR_HW_3~
WR_RD~
A2W_ENBL~
RD_HW_0~
RD_HW_1~
RD_HW_2~
RD_HW_3~
RD_WBUS~
LH
A2W_ENBL~ causes the Warp bus
to be driven from the address register:
HI - ALS652 A register to B port
LO - Address register to B port
Stored B register goes to A port


/home/white/symanski/iwarp/TB/4.drw 01 SEP 93 14:49 last update: 07 JAN 92 15:04
TB_(15:0)
Address Modes:
0 - Clear - Asynchronous
1 - Count Down
2 - Load
3 - Count Up
ADDR_MODE_0
ADDR_MODE_1
GND
74AS867
IC35
ADDR_RC1
LO_CLK~
LO_ADDR
WR_LO_ADDR~
74AS08
IC39
INC_ADDR~
IC36
MID_ADDR
ADDR_RC2
IC37
HI_ADDR
ADDR_RC3~
WR_HI_ADDR~
HI_CLK~
ADDR_(22:0)
74ALS245
IC33
LO_A2W
IC34
HI_A2W
LH
AD2WB
WR_CONTROL~
RD_CONTROL~
IC22
74AS10
AD~
BOTH WR_CONTROL~ and RD_CONTROL~ must be HI to avoid bus conflict. This is to enable clean writing and reading of the control bits.
74ALS652
IC38
PARITY_DATA
WR_PARITY~
A2W_ENBL~
WR_RD~
RD_WBUS~
RD_PARITY~
PDAT_0
PDAT_1
PDAT_2
PDAT_3
FM_DONE~
74AS04
IC1
J3 41
J3 9
J3 47
J3 107
J3 75
J3 128
J3 15
J3 48
J3 16
J3 87
J3 115
J3 83
J3 54
J3 22
J3 55
J3 25
J3 38
J3 29
J3 62
J3 61
J3 58
J3 98
J3 55
J3 108
J3 96
J3 129
J3 97


/home/white/symanski/iwarp/TB/5.drw 01 SEP 93 14:50 last update: 07 JAN 92 15:03
LDAT_0 J3 36
LDAT_1 J3 4
LDAT_2 J3 37
LDAT_3 J3 5
LDAT_4 J3 38
LDAT_5 J3 6
LDAT_6 J3 39
LDAT_7 J3 7
LDAT_8 J3 10
LDAT_9 J3 43
LDAT_10 J3 11
LDAT_11 J3 44
LDAT_12 J3 12
LDAT_13 J3 45
LDAT_14 J3 13
LDAT_15 J3 46
LDAT_(31:0)
LDAT_16 J3 49
LDAT_17 J3 17
LDAT_18 J3 50
LDAT_19 J3 18
LDAT_20 J3 51
LDAT_21 J3 19
LDAT_22 J3 52
LDAT_23 J3 20
LDAT_24 J3 23
LDAT_25 J3 56
LDAT_26 J3 24
LDAT_27 J3 57
LDAT_28 J3 26
LDAT_29 J3 59
LDAT_30 J3 27
LDAT_31 J3 60
HDAT_0 J3 102
HDAT_1 J3 70
HDAT_2 J3 103
HDAT_3 J3 71
HDAT_4 J3 104
HDAT_5 J3 72
HDAT_6 J3 105
HDAT_7 J3 73
HDAT_8 J3 76
HDAT_9 J3 109
HDAT_10 J3 77
HDAT_11 J3 110
HDAT_12 J3 80
HDAT_13 J3 113
HDAT_14 J3 81
HDAT_15 J3 114
HDAT_(31:0)
HDAT_16 J3 118
HDAT_17 J3 86
HDAT_18 J3 119
HDAT_19 J3 87
HDAT_20 J3 120
HDAT_21 J3 88
HDAT_22 J3 121
HDAT_23 J3 89
HDAT_24 J3 123
HDAT_25 J3 91
HDAT_26 J3 124
HDAT_27 J3 125
HDAT_28 J3 93
HDAT_29 J3 126
HDAT_30 J3 94
HDAT_31 J3 127
VCC
J1 B29
J1 B3
J2 D16
J1 B31
J1 B10
J1 B1
J2 D18
GND
NOTE: LEDs are ordered from the back of the board to the front, in bit order, as a hex number. The LSB is on the right, as viewed from the component side.
AD2WB
LED
LED1
QUAD_LED
RD_WR~
INC_ADDR
ADDR_MODE_1
ADDR_MODE_0
LED1
LED2
LED3
LED4
QLED2
QUAD_LED
MODE_1
MODE_0
HI_WORD
LO_WORD
LED1
LED2
LED3
LED4
QLED1
VCC
J3 92
J3 85
J3 78
J3 74
J3 69
J3 63
J3 47
J3 40
J3 34
J3 130
J3 122
J3 116
J3 112
J3 106
J3 99
J3 53
J3 31
J3 28
J3 21
J3 14
J3 8
J3 2
GND
NOTE: TPCs are ordered from the back of the board to the front.
RDY TPC24
EVENT TPC23
MCLK2 TPC22
16_MHZ TPC21
8_MHZ TPC20
TBRD~ TPC19
TBWR~ TPC18
TB~ TPC17
RD_CONTROL~ TPC16
EXECUTE~ TPC15
SW_RESET~ TPC14
RD_PARITY~ TPC13
RD_HW_3~ TPC12
RD_HW_2~ TPC11
RD_HW_1~ TPC10
RD_HW_0~ TPC9
WR_CONTROL~ TPC8
WR_HI_ADDR~ TPC7
WR_LO_ADDR~ TPC6
WR_PARITY~ TPC5
WR_HW_3~ TPC4
WR_HW_2~ TPC3
WR_HW_1~ TPC2
WR_HW_0~ TPC1


/home/white/symanski/iwarp/TB/6.dw 01 SEP 93 14:51 last update: 07 JAN 92 15:03
VCC
C01MFD
CCAP 1
CCAP 2
CCAP 3
CCAP 4
CCAP 5
CCAP 6
CCAP 7
CCAP 8
CCAP 9
CCAP 10
CCAP 11
CCAP 12
CCAP 13
CCAP 14
CCAP 15
CCAP 16
CCAP 17
CCAP 18
CCAP 19
CCAP 20
CCAP 21
CCAP 22
CCAP 23
CCAP 24
CCAP 25
CCAP 26
CCAP 27
CCAP 28
CCAP 29
CCAP 30
CCAP 31
CCAP 32
CCAP 33
CCAP 34
CCAP 35
CCAP 36
CCAP 37
CCAP 38
CCAP 39
CCAP 40
CCAP 41
CCAP 42
CCAP 43
CCAP 44
CCAP 45
CCAP 46
CCAP 47
CCAP 48
C56MFD
TCAP 1
TCAP 2
TCAP 3
TCAP 4
TCAP 5
TCAP 6
TCAP 7
TCAP 8
GND

Appendix B

TEST SOFTWARE

1 - ib.h - HEADER FILE FOR DISPLAY MODULE PROGRAMS

```
/* FILE: /home/white/symanski/iwarp/documents/report/ib.h        September 1993
*
* Header file for Image Board programs
*
* Author:   Jerry Symanski
*
*  The GAD must have bits 8 and 9 set to address CTLA = 0x060.
*       | 15 14 13 12 | 11 10  9  8 | 7  6  5  4 | 3  2  1  0 |
* GAD   |  0  0  0  0 |  0  0  1  1 | 0  0  0  0 | 0  x  x  x | GAD ignores x bits
*       |                   \---------/  \---------/          | 0x0060 = GAD address
* PTR   |  0  0  0  0 |  0  0  1  1 | 0  0  0  0 | 0  0  0  0 | 0x0300 = Byte address
*                     \-----------/
*
*****************************************************************************/
/*  VRAM pixels:  24bpp=1048576 - 16bpp=524288 - 8bpp=262144 */
#include <stdio.h>
#include <iwsys/getcfg.h>
#include <regnums.h>          /* Harish Nag */
#include <asm/gen_asm.h>      /* Harish Nag */
#include <ksupp/blink.h>      /* To control the QCB LEDs */
#include <ksupp/cs.h>
#include <ksupp/event.h>
static int *PMWR  = (int *)0x2000000; /* Function 8h=1000b - PM_RW_AB FAST WRITE */
static int *FCSW  = (int *)0x2400000; /* Function 9h=1001b - FAST CS WRITE       */
static int *PMRD  = (int *)0x2800000; /* Function Ah=1010b - PM_RW_AB FAST READ  */
static int *FCSR  = (int *)0x2c00000; /* Function Bh=1011b - FAST READ: Not used */
static int *VRAM  = (int *)0x3000000; /* Function Ch=1011b - VRAM base address */
static int *G364  = (int *)0x3400000; /* Function Dh=1011b - G364 base address */
static int *CSRG  = (int *)0x3800000; /* Function Eh=1011b - Slow Control read */
static int *RESET = (int *)0x3c00000; /* Function Fh=1011b - Software RESET */
static int *HALF_SYNC = (int *)0x3400108;   /* GAD 0x021 - SET TO:      15 */
static int *BACK_PRCH = (int *)0x3400110;   /* GAD 0x022 - SET TO:      50 */
static int *DISPLAY   = (int *)0x3400118;   /* GAD 0x023 - SET TO:     256 */
static int *SHRT_DISP = (int *)0x3400120;   /* GAD 0x024 - SET TO:      87 */
static int *BROAD_PLS = (int *)0x3400128;   /* GAD 0x025 - SET TO:     164 */
static int *V_SYNC    = (int *)0x3400130;   /* GAD 0x026 - SET TO:       6 */
static int *V_PRE_EQ  = (int *)0x3400138;   /* GAD 0x027 - SET TO:       2 */
static int *V_POST_EQ = (int *)0x3400140;   /* GAD 0x028 - SET TO:       2 */
static int *V_BLANK   = (int *)0x3400148;   /* GAD 0x029 - SET TO:      56 */
static int *V_DISPLAY = (int *)0x3400150;   /* GAD 0x02A - SET TO:    2048 */
static int *LINE_TIME = (int *)0x3400158;   /* GAD 0x02B - SET TO:     352 */
static int *LINE_STRT = (int *)0x3400160;   /* GAD 0x02C - SET TO:       0 */
static int *MEM_INIT  = (int *)0x3400168;   /* GAD 0x02D - SET TO:    2000 */
static int *TRAN_DLAY = (int *)0x3400170;   /* GAD 0x02E - SET TO:      48 */
static int *MASK_REG  = (int *)0x3400200;   /* GAD 0x040 - SET TO: ff ffff */
static int *MREG = (int *)0x3400200;        /* GAD 0x040 - SET TO:   ff ffff */
static int *CTLA = (int *)0x3400300;        /* GAD 0x060 - SET TO:   3C 3011 */
static int *CTLB = (int *)0x3400380;        /* GAD 0x070 - SET TO: FFFF FFFF */
static int *CURSOR_POSITION = (int *)0x3400638; /* GAD 0xc7 - Variable: +/- 4K */
static int *CURSOR_PALETTE  = (int *)0x3400508; /* GAD 0x0A1 to A3: 3 x 24-bpp lut */
static int *CLUT            = (int *)0x3400800; /* GAD 0x100 TO 0x1ff: Color LUT*/
static int *CURSOR_STORE    = (int *)0x3401000; /* GAD 0x200 to 3ff - 512 x 16 bits */
/*  COPY NEXT THREE LINES TO ALL PROGRAMS TO INITIALIZE LM_SIZE
          AND LOAD A NOP PARITY HANDLER                          */
/*
ENABLE_IMAGE_BOARD(); Initialize LM_SIZE - Load NOP parity handler - No event report
DISABLE_PARITY();
```

```
DISABLE_EVENT_RPT();
*/
/* FUNCTION: DISABLE_EVENT_RPT - Turn off bit 31 in event report enable register */
void  DISABLE_EVENT_RPT()
{
register int eventr;                                    /* XXX */
  /* turn off bit 31 in event report enable register */
  eventr = asm_readcsreg(CSR_EVENTR);                          /* XXX */
  asm_writecsreg(CSR_EVENTR, (eventr & 0x7FFFFFFF));           /* XXX */
} /* END OF DISABLE_EVENT_RPT */
/* FUNCTION: ENABLE_IMAGE_BOARD - Write LM_SIZE register to enable Image board */
void  ENABLE_IMAGE_BOARD()
{
/*   The RTS must be lied to in the config file so that it does not write
 *  into the Image board VRAM locations or the Image board control register.
 *  The config file specifies only normal fast RAM up to 0x07ffff.  This routine
 *  sets the LM_SIZE register enabling the cell with the image board to use
 *  the high memory locations without causing LM memory access errors.
 *
 *    The least significant byte sets how much memory is available in eight
 *  steps: bit 0 = 128K words, bit 1 = 256K words, ... bit 7 = 16 Megawords.
 *  All bits are set because we want to use the whole memory space.  The image
 *  board uses the top half of the memory space.
 *    The second byte sets how much FAST memory is available in eight
 *  steps: bit 0 = 128K words, bit 1 = 256K words, ... bit 7 = 16 Megawords
 *  Our iwarp has 512 megabytes which is 128 K words.
 */
asm_writecsreg(CSR_LMSIZE,  0x000001ff ); /* set LM_SIZE to enable image board */
} /* end of ENABLE_IMAGE_BOARD */
/* FUNCTION: DISABLE_PARITY - This function disables the lm parity event.
 * It should ONLY be called by the cell with the frame buffer.
 * It disables parity by loading a no-op parity event handler.
 *
 * Written by William Shubert of Intel.
 */
void  DISABLE_PARITY()
{
 static unsigned  handler[] = {0x0e40005e,          /* ldlithz 0x8000,ev0 */
                               0x00a01dde,  /* movecsr ev0,eventc */
                               0x11ce0000}; /* retmfe             */
 unsigned int  **xbase;
 xbase = (unsigned int **)asm_readcsreg(CSR_XBASE);
 xbase[31] = handler;
 xbase[63] = handler;
} /* end of DISABLE_PARITY */
/* FUNCTION: CLEAR_DISPLAY - Write zeroes to all VRAM locations.         */
void  CLEAR_DISPLAY()
{
int pixel=0, p=0;
  for ( p=0; p<1048576; p++ ) /* 24bpp=1048576 - 16bpp=524288 - 8bpp=262144 */
    {
    FCSW[0] = 0x00;     /* enable G364 - clear bit 2 */
    VRAM[ p ] = pixel;
    } /* end of for p */
} /* end of CLEAR_DISPLAY */
/* FUNCTION: LOAD_DISPLAY - Write an 8-bit color to all VRAM locations.    */
void  LOAD_DISPLAY( color )
int color;
{
```

```
int  p=0;
  color = ( color & 0x00ff );
  color = ( ( color <<  8 ) | color );
  color = ( ( color << 16 ) | color );
  for ( p=0; p<1048576; p++ ) /* 24bpp=1048576 - 16bpp=524288 - 8bpp=262144 */
    {
    FCSW[0] = 0x00;     /* enable G364 - clear bit 2 */
    VRAM[ p ] = color;
    } /* end of for p */
} /* end of CLEAR_DISPLAY */
/* FUNCTION: RAMP - Write a one raster line ramp for the 1024 pixel display  */
void  RAMP()
{
unsigned int row, col, pixel=0, start;

  for ( row=0; row<4096; row++ )
    {
    start = (row * 256);
    for (col=0; col<256; col++ )
      {
      pixel = ( col );
      pixel = ( ( pixel <<  8 ) | pixel );
      pixel = ( ( pixel << 16 ) | pixel );
      FCSW[0] = 0x00;
      VRAM[ start + (col)] = pixel;
      } /* end of col */
    } /* end of row */
} /* end of RAMP */
/* FUNCTION: LOAD_24BPP - Setup the G364 for 24 bit-per-pixel display.         */
void  LOAD_24BPP()
{
int n=0, lut=0, val=0, data=0, k=0;
asm_writecsreg(CSR_LMSIZE,  0x000001ff );
/* set LM_SIZE register to enable image board */
  RESET[0] = 0;               /* software reset the IB */
  for ( n=0; n<50; n++ ) ;    /* wait 50 microseconds  */
  FCSW[0]  = 0x04;            /* RESET the G364        */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  FCSW[0]  = 0x00;            /* ENABLE the G364       */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  G364[0]  = 0x69;            /*  set PLL - 0x 69 = 90 MHz */
  for ( n=0; n<40; n++ ) ;    /* wait 40 microseconds  */
  CTLA[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLB[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  HALF_SYNC[0] =   15;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BACK_PRCH[0] =   50;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  DISPLAY  [0] =  256;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  SHRT_DISP[0] =   87;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BROAD_PLS[0] =  164;    /* GAD 0x025 - SET TO:     164 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_SYNC   [0] =    6;    /* GAD 0x026 - SET TO:       6 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_PRE_EQ [0] =    2;    /* GAD 0x027 - SET TO:       2 */
```

```
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_POST_EQ[0] =    2;    /* GAD 0x028 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_BLANK  [0] =   56;    /* GAD 0x029 - SET TO:      56 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_DISPLAY[0] = 2048;    /* GAD 0x02A - SET TO:    2048 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_TIME[0] =  352;    /* GAD 0x02B - SET TO:     352 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_STRT[0] =    0;    /* GAD 0x02C - SET TO:       0 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MEM_INIT [0] = 480;    /* GAD 0x02D - SET TO:    2000 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  TRAN_DLAY[0] =   32;    /* GAD 0x02E - SET TO:      48 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MASK_REG [0] = 0x00ffffff;    /* GAD 0x02E - SET TO: 00ffffff */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLA[0] = 0x00ec3011;     /* bc3011=8bpp - ec3011=24bpp */
/*  See the Inmos IMS G364 colour video controller manual, page 42, for
*    complete information on the use of the control register.
*   The "ec 3011" sets up 24 bit per pixel mode, cursor disabled.
*         || ||||--- Bit 0 enables the display.
*         || |||---- Plain composite sync - composite video + sync - no blank pedestal.
*         || ||----- Blanking.
*         || |------ Non-interlace increment is 1024.  The "3" must be used because
*         ||          of an error in the design.  The G364 address lines are incorrect
*         ||          and the VRAM address increment must be 1024 instead of 512, which
*         ||          is the VRAM row size. ( See page 24 of the Inmos manual. )
*         ||------- The "c" selects interleaved mode and enables delayed sampling.
*         |-------- The "e" selects 24 bits per pixel and disables the cursor.
*
*/
  for ( n=0; n<10; n++ ) ; /* wait 10 microseconds  */
  for ( lut=0; lut<512; lut=lut+2 )
    {                          /* inc by 2 since LUT is lower int only */
    val = (lut>>1);           /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ lut ] = data;       /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
}/* end of LOAD_24BPP  */
/* FUNCTION: LOAD_16BPP - Setup the G364 for 16 bit-per-pixel display */
void  LOAD_16BPP()
{
int n=0, lut=0, val=0, data=0, k=0;
asm_writecsreg(CSR_LMSIZE,   0x000001ff );
/* set LM_SIZE register to enable image board */
  RESET[0] = 0;              /* software reset the IB */
  for ( n=0; n<50; n++ ) ;    /* wait 50 microseconds  */
  FCSW[0]  = 0x04;           /* RESET the G364        */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  FCSW[0]  = 0x00;           /* ENABLE the G364       */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  G364[0]  = 0x69;         /* set PLL - 0x 69 = 90 MHz */
  for ( n=0; n<40; n++ ) ;    /* wait 40 microseconds  */
  CTLA[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLB[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
```

```
  HALF_SYNC[0] =   15;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BACK_PRCH[0] =   50;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  DISPLAY  [0] =  256;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  SHRT_DISP[0] =   87;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BROAD_PLS[0] =  164;    /* GAD 0x025 - SET TO:     164 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_SYNC   [0] =    6;    /* GAD 0x026 - SET TO:       6 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_PRE_EQ [0] =    2;    /* GAD 0x027 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_POST_EQ[0] =    2;    /* GAD 0x028 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_BLANK  [0] =   56;    /* GAD 0x029 - SET TO:      56 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_DISPLAY[0] = 2048;    /* GAD 0x02A - SET TO:    2048 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_TIME[0] =  352;    /* GAD 0x02B - SET TO:     352 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_STRT[0] =    0;    /* GAD 0x02C - SET TO:       0 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MEM_INIT [0] =  992;    /* GAD 0x02D - SET TO:     992 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  TRAN_DLAY[0] =   32;    /* GAD 0x02E - SET TO:      32 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MASK_REG [0] = 0x00ffffff;   /* GAD 0x02E - SET TO: 00ffffff */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLA[0] = 0x00dc3011;     /* bc3011=8bpp - dc3011=16bpp - ec3011=24bpp */
                            /*  bit 23=1 to DISABLE the cursor */
  for ( n=0; n<10; n++ ) ; /* wait 10 microseconds  */
  for ( lut=0; lut<512; lut=lut+2 )
    {                              /* inc by 2 since LUT is lower int only */
    val = (lut>>1);               /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ lut ] = data;       /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
}/* end of LOAD_16BPP  */
/* FUNCTION: LOAD_8BPP - Setup the G364 for 8 bit-per-pixel display.  */
void  LOAD_8BPP()
{
int n=0, lut=0, val=0, data=0, k=0;
asm_writecsreg(CSR_LMSIZE,   0x000001ff );
/* set LM_SIZE register to enable image board */
  RESET[0] = 0;               /* software reset the IB */
  for ( n=0; n<50; n++ ) ;    /* wait 50 microseconds  */
  FCSW[0]  = 0x04;            /* RESET the G364        */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  FCSW[0]  = 0x00;            /* ENABLE the G364       */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  G364[0]  = 0x69;          /*  set PLL - 0x 69 = 90 MHz */
  for ( n=0; n<40; n++ ) ;    /* wait 40 microseconds  */
  CTLA[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLB[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
```

```
  HALF_SYNC[0] =   15;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BACK_PRCH[0] =   50;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  DISPLAY  [0] =  256;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  SHRT_DISP[0] =   87;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BROAD_PLS[0] =  164;    /* GAD 0x025 - SET TO:     164 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_SYNC   [0] =    6;    /* GAD 0x026 - SET TO:       6 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_PRE_EQ [0] =    2;    /* GAD 0x027 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_POST_EQ[0] =    2;    /* GAD 0x028 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_BLANK  [0] =   56;    /* GAD 0x029 - SET TO:      56 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_DISPLAY[0] = 2048;    /* GAD 0x02A - SET TO:    2048 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_TIME[0] =  352;    /* GAD 0x02B - SET TO:     352 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_STRT[0] =    0;    /* GAD 0x02C - SET TO:       0 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MEM_INIT [0] = 2000;    /* GAD 0x02D - SET TO:    2000 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  TRAN_DLAY[0] =   48;    /* GAD 0x02E - SET TO:      48 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MASK_REG [0] = 0x00ffffff;    /* GAD 0x02E - SET TO: 00ffffff */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLA[0] = 0x00bc3011;     /* bc3011=8bpp - dc3011=16bpp - ec3011=24bpp */
                            /* bit 23=1 to DISABLE the cursor */
  for ( n=0; n<10; n++ ) ; /* wait 10 microseconds  */
  for ( lut=0; lut<512; lut=lut+2 )
    {                              /* inc by 2 since LUT is lower int only */
    val = (lut>>1);               /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ lut ] = data;       /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
}/* end of LOAD_8BPP  */
/* FUNCTION: LOAD_4BPP - Setup the G364 for 8 bit-per-pixel display.  */
void  LOAD_4BPP()
{
int n=0, lut=0, val=0, data=0, k=0;
asm_writecsreg(CSR_LMSIZE,   0x000001ff );
/* set LM_SIZE register to enable image board */
  RESET[0] = 0;               /* software reset the IB */
  for ( n=0; n<50; n++ ) ;    /* wait 50 microseconds  */
  FCSW[0]  = 0x04;            /* RESET the G364        */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  FCSW[0]  = 0x00;            /* ENABLE the G364       */
  for ( n=0; n<25; n++ ) ;    /* wait 25 microseconds  */
  G364[0]  = 0x69;          /* set PLL - 0x 69 = 90 MHz */
  for ( n=0; n<40; n++ ) ;    /* wait 40 microseconds  */
  CTLA[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLB[0] = 0x00000000;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
```

```
  HALF_SYNC[0] =   15;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BACK_PRCH[0] =   50;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  DISPLAY  [0] =  256;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  SHRT_DISP[0] =   87;
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  BROAD_PLS[0] =  164;    /* GAD 0x025 - SET TO:     164 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_SYNC   [0] =    6;    /* GAD 0x026 - SET TO:       6 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_PRE_EQ [0] =    2;    /* GAD 0x027 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_POST_EQ[0] =    2;    /* GAD 0x028 - SET TO:       2 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_BLANK  [0] =   56;    /* GAD 0x029 - SET TO:      56 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  V_DISPLAY[0] = 2048;    /* GAD 0x02A - SET TO:    2048 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_TIME[0] =  352;    /* GAD 0x02B - SET TO:     352 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  LINE_STRT[0] =    0;    /* GAD 0x02C - SET TO:       0 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MEM_INIT [0] = 4048;    /* GAD 0x02D - SET TO:    2000 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  TRAN_DLAY[0] =   48;    /* GAD 0x02E - SET TO:      48 */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  MASK_REG [0] = 0x00ffffff;    /* GAD 0x02E - SET TO: 00ffffff */
  for ( n=0; n<10; n++ ) ;    /* wait 10 microseconds  */
  CTLA[0] = 0x00ac3011;     /* ac=4bpp - bc=8bpp - dc=16bpp - ec=24bpp */
                            /* bit 23=1 to DISABLE the cursor */
  for ( n=0; n<10; n++ ) ; /* wait 10 microseconds  */
  for ( lut=0; lut<512; lut=lut+2 )
    {                              /* inc by 2 since LUT is lower int only */
    val = (lut>>1);                /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ lut ] = data;            /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
}/* end of LOAD_4BPP  */
/* FUNCTION: LOAD_CHECKERS - Write a checker board pattern for 24 BPP mode  */
void LOAD_CHECKERS( red, green, blue )
int red, green, blue;
{
int pixel=0, p=0;
  pixel = ( ( red << 16 ) + ( green<<8) + blue );      /* build rgb integer */
  for ( p=0; p<1048576; p++ )                          /* 24bpp = 1048576 words */
    {
    if ( p%32==0 )    pixel = ( pixel ^ 0x00ffffff ); /* horizontal blocks */
    if ( p%32768==0 ) pixel = ( pixel ^ 0x00ffffff ); /* vertical blocks */
    FCSW[0] = 0x00;                                   /* enable G364 - clear bit 2 */
    VRAM[ p ] = pixel;
    } /* end of for p */
} /* end of LOAD_CHECKERS */
/* FUNCTION: LOAD_CHECKERS8 - Write a checker board pattern for 8 BPP mode  */
void LOAD_CHECKERS8( pixel )
unsigned int  pixel;
{
```

```
int  p=0;
  pixel = ( pixel & 0x00ff );          /* build integer with four pixels */
  pixel = ( ( pixel <<  8) | pixel ); /* fill lower two bytes - lower half int */
  pixel = ( ( pixel << 16) | pixel ); /* fill upper two bytes - upper half integer */
  for ( p=0; p<262144; p++ )          /* 24bpp=1048576 - 16bpp=524288 - 8bpp=262144 */
    {
    if ( p%32==0 )    pixel = ( pixel ^ 0xffffffff );     /* horizontal blocks */
    if ( p%32768==0 ) pixel = ( pixel ^ 0xffffffff );     /* vertical blocks */
    FCSW[0] = 0x00;                                       /* enable G364 - bit 2 */
    VRAM[p] = pixel;
    } /* end of for p */
} /* end of LOAD_CHECKERS8*/
/* FUNCTION: LOAD_LUT8 - Load the G364 Color LUT for 8 bit-per-pixel display
*
*  This color look-up table goes from black to red, yellow, orange, green, blue,
*  violet to white in seven sections.  It is strictly a mathematical generation.
*  No physiological basis was used.
*
*  Note that red is in the least significant byte of the 24 bit LUT word.
*  Ie.,  blue-green-red.
*
*/
void  LOAD_LUT8()
{
unsigned  int  red, grn, blue, lut[256], val=0, data=0, lut_data[512];
int n=0, k=0;
  for ( n=0; n<=31; n++ )  /* black at 0 to pure red at 32 */
    {
    blue = ( n * 8 );  grn = 0;  red = 0;
    lut[n] = ( (blue << 16 ) | (grn << 8 ) | ( red ) );
    } /* end of first 32 */
  for ( n=32; n<=63; n++ )  /* red at 32 to pure yellow at 63 */
    {
    blue = 255;  grn = ( ( 8 * ( n-31 ) ) - 1 );  red = 0;
    lut[n] = ( (blue << 16 ) | (grn << 8 ) | ( red ) );
    } /* end of first 64 */
  for ( n=64; n<=95; n++ )  /* pure yellow at 64 to pure green at 95 */
    {
    blue = ( 255 - ( 8 * ( n-64) ) );  grn = 255;  red = 0;
    lut[n] = ( (blue << 16 ) | ( grn << 8 ) | ( red ) );
    } /* end of first 96 */
  for ( n=96; n<=159; n++ )  /* turquoise at 159 */
    {
    blue = 0;  grn = ( 255 - ( 4 * ( n-96 ) ) );
    red = ( ( ( n-95 ) * 4 ) - 1 );
    lut[n] = ( (blue << 16 ) | (grn << 8 ) | ( red ) );
    } /* end of first 160 */
  for ( n=160; n<=223; n++ )  /* pure blue at 160 - violet at 223 */
    {
    blue = ( ( ( n - 159 ) * 4 ) - 1 );  grn = 0;  red = 255;
    lut[n] = ( (blue << 16 ) | (grn << 8 ) | ( red ) );
    } /* end of first 224 */
  for ( n=224; n<=255; n++ )  /* violet at 224 - white at 256 */
    {
    blue = 255;  grn = ( ( ( n-223 ) * 8 ) -1);  red = 255;
    lut[n] = ( (blue << 16 ) | (grn << 8 ) | ( red ) );
    } /* end of lut */
  for ( n=0; n<512; n=n+2 )  /* LOAD CLUT */
    {                        /* inc by 2 since LUT is lower int only */
```

```
    val = (n>>1);              /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ n ] = lut[(n>>1)];       /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
} /* end of LOAD_LUT8 */
/* FUNCTION: LOAD_LUT24 - Load the G364 Color LUT for 24 bit-per-pixel display
*
*   This function loads the LUT with a ramp going from 0 to 255.  Thus the
*  three bytes of the 24 bit color integer will be interpreted just as they
*  are.  This is a one-to-one mapping.
*
*/
void  LOAD_LUT24()
{
unsigned  int    lut=0, val=0, data=0;
int n=0, k=0;
  for ( lut=0; lut<512; lut=lut+2 )
    {                              /* inc by 2 since LUT is lower int only */
    val = (lut>>1);               /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ lut ] = data;        /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
} /* end of LOAD_LUT24 */
/* FUNCTION: LOAD_LUT4 - Load the G364 Color LUT for 8 bit-per-pixel display */
void  LOAD_LUT4()
{
unsigned  int   red, grn, blue, lut[256], val=0, data=0, lut_data[512];
int n=0, k=0;
  for ( n=0; n<512; n=n+2 )  /* LOAD CLUT */
    {                           /* inc by 2 since LUT is lower int only */
    val = (n>>1);               /* divide by 2 since lut is double */
    data = ( (val<<16) | (val<<8) | val );
    CLUT[ n ] = lut[(n>>1)];       /* load through lut */
    for ( k=0; k<5; k++ ) ;
    }/* end of for lut */
    CLUT[  0 ] = 0x00000000;  /* load the first 16 locations for 4 bpp */
    CLUT[  2 ] = 0x00800000;
    CLUT[  4 ] = 0x00ff0000;
    CLUT[  6 ] = 0x00fff000;
    CLUT[  8 ] = 0x00ffff00;
    CLUT[ 10 ] = 0x007fff00;
    CLUT[ 12 ] = 0x0000ff00;
    CLUT[ 14 ] = 0x00007f00;
    CLUT[ 16 ] = 0x0000007f;
    CLUT[ 18 ] = 0x000000ff;
    CLUT[ 20 ] = 0x00000080;
    CLUT[ 22 ] = 0x00000040;
    CLUT[ 24 ] = 0x00040040;
    CLUT[ 26 ] = 0x00800080;
    CLUT[ 28 ] = 0x00ff00ff;
    CLUT[ 30 ] = 0x00ffffff;
} /* end of LOAD_LUT4 */
/* FUNCTION: disable_lm_parity - Disable the LM parity reporting  */
disable_lm_parity()
{
asm_writecsreg(CSR_EVENTR, asm_readcsreg(CSR_EVENTR) & 0x7FFFFFFF);
}
```

```
/* FUNCTION: enable_lm_parity - Enable the LM parity reporting    */
enable_lm_parity()
{
asm_writecsreg(CSR_EVENTR, asm_readcsreg(CSR_EVENTR) | 0x80000000);
}
/* FUNCTION: INIT_IB - Do ENABLE_IMAGE_BOARD and DISABLE_PARITY */
void  INIT_IB()
{
ENABLE_IMAGE_BOARD();
DISABLE_PARITY();
LOAD_8BPP();
LOAD_CHECKERS8( 0 );
} /* end of INIT_IB */
```

2 - vram.c - TEST VRAM WRITING

```
/* File: ~/symanski/iwarp/documents/report/vram.c
*
* Test the VRAM of the image display board.
*
* Author:   Jerry Symanski with added XXX code by Harish Nag of Intel
* History:  8 September 1993
*
*******************************************************************************/
#include "ib.h"
#define LOOPS 100000
#define LMOD   100
main()
{
int  vdata=0, wdata=0, loops=0;
int  l=0, i=0, k=0, n=0, errn=0, adr=0;
int  loc[256], err[256];
register int eventr;                                /* XXX */
struct iwcfg cfg;
getcfg (&cfg);
fprintf(stderr,  "vram:  Starting  in  cell  %2d    -    %4d  loops:\n",  cfg.cellid,  LOOPS  );
fflush(stderr);
 ENABLE_IMAGE_BOARD();
 DISABLE_PARITY();
 LOAD_24BPP();
 LOAD_CHECKERS( 128, 32, 64 );
  for ( l=0; l<256; l++ ) { loc[l]=0; err[l]=0; } /* clear arrays */
  for ( loops=0; loops<LOOPS; loops++ )
    {
    for ( adr=0; adr<1048576; adr++ )
      {
      wdata = ( rand() << ( loops & 0xf ) );
      VRAM[adr] = wdata;       /* WRITE = 450 nanosec */
      FCSW[0] = 0x08;          /* WRITE to CS to setup addresses */
      vdata = VRAM[adr];
      FCSW[0] = 0x00;          /* WRITE to CS to setup addresses */
      if ( ( vdata != wdata ) && ( errn <=255 ) )
        {
        err[errn] = (   wdata );
        loc[errn] = adr;
        errn++;
        }
      } /* end of for all adr */
    if ( loops%LMOD == 0 )
      {
       fprintf(stderr, "vram: Did %4d loops.  Errors: %d\n", loops, errn ); fflush(stderr);
      }
    } /* end of for loops */
  fprintf(stderr, "vram is done.    Errors: %8d\n", errn ); fflush(stderr);
  if ( errn>0 )
    for ( n=0; n<10; n++ )
      printf("Error Number: %8d  X: %8x  Loc: %8d\n", n, err[n], loc[n]);
  exit (0);
} /* end of vram */
```

3 - testg3.c - TEST G364 VIDEO CONTROLLER

```
/*  File: ~symanski/iwarp/report/testg3.c              September 1993
*
*  Test program to read the G364 control registers. Note that the G364 must be
* disabled to read the registers.
**
* Author:   Jerry Symanski with Disable of Parity by Bill Shubert of intel
* History:  25 Feb 1993
*
*******************************************************************************/
#include "ib.h"
main()
{
struct iwcfg cfg;
int gdata=0, n=0;

getcfg (&cfg);
printf ("testg3: Cell #%d: \n", cfg.cellid );
ENABLE_IMAGE_BOARD();
DISABLE_PARITY();
LOAD_8BPP();   LOAD_LUT8();
LOAD_CHECKERS8( 64 );
gdata = CTLA[0];
printf("CONTROL BEFORE:            %6x\n",   (gdata & 0x00ffffff) ); /* OK */
CTLA[0] = 0x00bc3010;
for ( n=0; n<10; n++ ) ; /* MUST wait 10 microseconds after messing with VTG */
gdata = CTLA[0];
for ( n=0; n<10; n++ ) ; /* MUST wait 10 microseconds after messing with VTG */
printf("CONTROL AFTER:             %6x\n",   (gdata & 0x00ffffff) );
gdata = HALF_SYNC[0];       printf("HALF_SYNC:       [    15]  %6d\n",   (gdata & 0x0000ffff) );
gdata = BACK_PRCH[0];       printf("BACK_PORCH:      [    50]  %6d\n",   (gdata & 0x0000ffff) );
gdata = DISPLAY[0];         printf("DISPLAY:         [   256]  %6d\n",   (gdata & 0x0000ffff) );
gdata = SHRT_DISP[0];       printf("SHORT_DISPLAY:   [    87]  %6d\n",   (gdata & 0x0000ffff) );
gdata = BROAD_PLS[0];       printf("BROAD_PULSE:     [   164]  %6d\n",   (gdata & 0x0000ffff) );
gdata = V_SYNC[0];          printf("V_SYNC:          [     6]  %6d\n",   (gdata & 0x0000ffff) );
gdata = V_PRE_EQ[0];        printf("V_PRE_EQ:        [     2]  %6d\n",   (gdata & 0x0000ffff) );
gdata = V_POST_EQ[0];       printf("V_POST_EQ:       [     2]  %6d\n",   (gdata & 0x0000ffff) );
gdata = V_BLANK[0];         printf("V_BLANK:         [    56]  %6d\n",   (gdata & 0x0000ffff) );
gdata = V_DISPLAY[0];       printf("V_DISPLAY:       [  2048]  %6d\n",   (gdata & 0x0000ffff) );
gdata = LINE_TIME[0];       printf("LINE_TIME:       [   352]  %6d\n",   (gdata & 0x0000ffff) );
gdata = LINE_STRT[0];       printf("LINE_START:      [     0]  %6d\n",   (gdata & 0x0000ffff) );
gdata = MEM_INIT[0];        printf("MEM_INIT:        [  2000]  %6d\n",   (gdata & 0x0000ffff) );
gdata = TRAN_DLAY[0];       printf("TRANSFER_DELAY:  [    48]  %6d\n",   (gdata & 0x0000ffff) );
MASK_REG[0] = 0x0000fffff;
gdata = MASK_REG[0];        printf("MASK_REG:        [ffffff]  %06Xh\n", (gdata & 0x00ffffff) );
gdata = CTLA[0];            printf("CONTROL_A:       [BC3011]  %06Xh\n", (gdata & 0x00ffffff) );
for ( n=0; n<10; n++ ) ; /* MUST wait 10 microseconds after messing with VTG */
CTLB[0] = 0x00000000;
for ( n=0; n<10; n++ ) ; /* MUST wait 10 microseconds after messing with VTG */
gdata = CTLB[0];            printf("CONTROL_B:       [000000]  %06Xh\n", (gdata & 0x00ffffff) );
CURSOR_POSITION[0] = 0x000000;
gdata = CURSOR_POSITION[0];
printf("CURSOR_POSITION:[000000]   %06Xh\n", (gdata & 0x00ffffff) );
LOAD_CHECKERS8( 100 );
CTLA[0] = 0x00bc3011;
for ( n=0; n<10; n++ ) ; /* MUST wait 10 microseconds after messint with VTG */
gdata = CTLA[0];
printf("CONTROL NOW:               %6Xh\n",  (gdata & 0x00ffffff) );
```

```
printf ("testg3 is done. \n");
exit (0);
} /* end of testg3
For the 8 BPP display:
do testg3
Loading iwarp with testg3
SIB 0 on teal has been locked.
testg3 has finished....
SIB 0 on teal has been unlocked.
testg3: Cell #21:
CONTROL BEFORE:                bc3011
CONTROL AFTER:                 bc3010
HALF_SYNC:          [    15]       15
BACK_PORCH:         [    50]       50
DISPLAY:            [   256]      256
SHORT_DISPLAY:      [    87]       87
BROAD_PULSE:        [   164]      164
V_SYNC:             [     6]        6
V_PRE_EQ:           [     2]        2
V_POST_EQ:          [     2]        2
V_BLANK:            [    56]       56
V_DISPLAY:          [  2048]     2048
LINE_TIME:          [   352]      352
LINE_START:         [     0]        0
MEM_INIT:           [  2000]     2000
TRANSFER_DELAY:     [    48]       48
MASK_REG:          [ffffff]   0FFFFFh
CONTROL_A:         [BC3011]   BC3010h
CONTROL_B:         [000000]   000000h
CURSOR_POSITION:[000000]      000000h
CONTROL NOW:                   BC3011h
testg3 is done.
 */
```

4 - maxf512.c - TEST EVENT AND PAGE MODE OPERATION

```
/*   File:   ~symanski/iwarp/documents/report/maxf512.c
 *
 *     Test writing a buffer into the center 512x512 window of
 *   the 1024x1024 8 BPP display.  This code uses the Serial Access Memory
 *   Transfer(SAMT) event with John Webb's asm_copy_64 routine.
 *
 *   This program wirtes double words into VRAM at 250 nanoseconds per write.
 *   The efficiency is about 62%.  The viewed frames per second is about 60 fps.
 *
 *   Peak data rate is 32 MBytes per second or 128 512x512 frames/sec.
 *   Efficiency could be improved with a more clever event handler.
 *
 *   Note that care must be taken to catch every event request.  Too long a
 *   write period, will cause some requests to be missed, lowering efficiency.
 *
 *   Note: Can not do I/O from event handler.  26244
 *
  ********************************************************************************/
#include "ib.h"
#include "asm_copy_64.h"
#define LOOPS    400000
#define FRAMES     8192
void  vram_write();
struct iwcfg  cfg;
unsigned int PAGE=32, LINE=0, FRAME=0, vram_page, now=0, old=0;
double  *DPMWR = (double *)0x2000000; /* Page mode VRAM base address */
double  dbuf[1024];
static union   { double dwd; int wd[2]; } img;
main()
{
unsigned int   n=0, j=0, pixel=0, dummy=0, loop=0, mult=0, pix_adr=0;
ENABLE_IMAGE_BOARD();
DISABLE_PARITY();
LOAD_8BPP();    LOAD_LUT8();                  /* initialize the graphics chip */
LOAD_CHECKERS8( 20 );           /* load a checker board to verify operation */
LOAD_DISPLAY( 100 );
img.wd[0]=0xc0c0c0c0; img.wd[1]=0xc0c0c0c0; /* load a double word with color */
for ( n=0; n<1024; n++ )  dbuf[n] = ( img.dwd ); /* load dbuf with the color */
fprintf(stderr, "Starting maxf512:  %9d frames.\n", (LOOPS*FRAMES) ); fflush(stderr);
/* Install the handler, then enable the event */
install_handler(EVENT_EXTERNAL, vram_write, 0, EVH_LOCALE_C);
for ( loop=1; loop<LOOPS; loop++)
{
mask_event(CSR_EVENTR, 1<<EVENT_EXTERNAL, 1<<EVENT_EXTERNAL); /* enable the event */
CSRG[0] = 0x02;     /* enable event signal */
while ( FRAME<FRAMES )
  {
  now = FRAME;
  if ( now != old )
    {
    pixel = ( FRAME & 0x00ff );
    pixel = ( ( pixel<<24 ) | ( pixel<<16 ) | ( pixel<<8 ) | pixel );
    img.wd[0]=pixel;  img.wd[1]=pixel;
    for ( j=0; j<128; j++ ) dbuf[j] = img.dwd;
    old = now;
    }
  pix_adr = ( (FRAME+((loop-1)*FRAMES)) & 0x003ffff );
```

```
  VRAM[pix_adr] = 0xffffffff;/* draw comet */
  }
CSRG[0] = 0;  /* disable the event signal */
/* Disable the event before terminating the program or printing out. */
mask_event(CSR_EVENTR, 1<<EVENT_EXTERNAL, 0); /* disable the event process. */
fprintf(stderr, "maxf512: %6d frames: loop=%6d \n", (loop*FRAME), loop );
fflush(stderr);
FRAME = 0;
if ( loop%32==0 ) LOAD_DISPLAY( loop&0x00ff );
} /* end of loop */
printf("maxf512 is done.   Did %6d frames.\n", (loop*FRAME) );
exit(0);
} /* end of main of maxf512 */
£
/* EVENT_HANDLER:  vram_write - Respond to Graphics controller signal
 *
 *    This event handler will write pixels to the Image board VRAM as fast as
 *  possible.  The event will be activated with a period depending on the
 *  pixel depth.  When using 8 bits-per-pixel(bpp), the event will be triggered
 *  every 128 microseconds by the rise of a 2 microsecond wide signal.  When
 *  the 2 microseconds is finished, the graphics controller will be done
 *  with the VRAM address bus and the iWarp can take control of the VRAM
 *  for the next 125 microseconds, writing at the maximum 100 nanosecond
 *  per 64 bit word rate if possible. For 16 bits-per-pixel, signal
 *  occurs every 64 microseconds.
 *
 *    This routine will use the page-mode of writing into the VRAM. Ie.,
 *  the VRAM RAS signal will go high only once, at which time it latches
 *  the row address into the VRAM.  Then,  writes can be performed for
 *  approximately the next 120 microseconds, in the 8 bpp mode.  If the 100
 *  nanosecond period cannot be achieved, this routine will have to stop
 *  writing after 120 microseconds and relinquish control to the graphics
 *  controller so that new image data can be loaded into the VRAM serial
 *  register.  Unless the graphics controller can get control every 128
 *  microseconds, the display will be noisy and corrupted.
 *
 *    The pixels can be transfered from an input buffer dbuf, to the VRAM.
 *  The maximum number of writes is 1024 64 bit words.  The Image board
 *  has two banks of VRAM, each having 512 locations per row.  Fewer writes
 *  are acceptable if the number of writes is saved to that the location
 *  for the next data to be written is available.
 *
 */
void  vram_write(ev_num, pct_num, dummy, eventr)
int  ev_num, pct_num, dummy, *eventr;
{
unsigned int  col=0, start=0, blk=0;
vram_page = ( (  PAGE << 11 ) ); /* generate page address ie., VRAM row */

CSRG[vram_page] = 0x01;    /* load page into VRAM with RAS - clear the event bit */
start = ( LINE * 128 );                    /* compute current raster line to load */
/*
 Because of the write cycle length and the available time between G364 SAMT requests,
when doing 1Kx1K screens, 128 words = 1024 bytes is written in each LINE.
A PAGE in the VRAM contains 2x512=1024 double integers = 8096 bytes.
So to write a total of 8096 bytes, there were 8 block writes of 128*8=1024 bytes. But
these block writes could only be done three at a time to fit inside the 120 microseconds
between SAMT requests. The code for a 1024x1024 display does a 3 x 128 double word write,
another 3 x 128 double word write and then a 2 x 128 double word write.  This results in
```

```
about 20 frames per second for the 1024x1024 8-bit per pixel display.
For the 512x512 display, we can fit four 64 double word writes during each page mode write.
This results in writing for approximately 78 microseconds out of the 125 microseconds
available for a 62% efficiency.  This is is about 60 512x512 frames per second.
With more assembly code, efficiency could probably be brought up to 90%, or over 100
512x512 frames per second.
*/
for ( blk=0; blk<4; blk++ )
  {
  copy_words(64, &dbuf[0], &DPMWR[(blk*128) + start + 32 ], sizeof(double), sizeof(double));
  LINE = LINE + 1;
  if ( LINE == 8 ) { LINE = 0; PAGE = PAGE + 1; goto DONE; } /* no 9th LINE */
  }
DONE:
FCSW[0] = 0x02;                                    /* clr PM bit - enable event */
if ( PAGE>96 ) { PAGE=32; FRAME++; }                    /* check for frame done */
asm_writecsreg(CSR_EVENTC, 1<<EVENT_EXTERNAL);            /* Disable the event */
} /* end of vram_write */
```

5 - master.c, frame.c, frame.ad, fastio.h - TEST ADAPT USE OF THE DISPLAY MODULE

```
/* FILE: ~symanski/iwarp/documents/report/master.c.add_one_bw
*
*    Uses stdin0064 for 512x512 image.
*
*    master.c from Jon Webb with additions by Symanski to drive the Sony monitor.
*
*    This program writes frames at about 18 FPS to a 512x512 window centered
*    in the the 1kx1k screen. It uses the ib_receive_words which does a single
*    word write to FCSW to guarantee an addressing transition.  Writes take
*    about 650 nanoseconds each.
*
*/
#include <stdio.h>
#include <netcode.h>
#define HEIGHT 512
#define WIDTH  512
#define MAXSIZE 4096
#define IMG_SIZE 262144
#define FRAMES 1000000
#define PCS_ERROR { fprintf(stderr, "PCS Error in file %s line %d\n", __FILE__,__LINE__);\
                        pcs_fatal(NULL); }
main()
{
int img_id, res_id, frame;
char  img_bufr[IMG_SIZE];
FILE *input_image;
   fprintf(stderr, "master.c.add_one_bw:  \nStarting initialization\n");    fflush(stderr);
   Initialize_Adapt();
   fprintf(stderr, "Finished initialization\n");    fflush(stderr);
   read_input(0, img_bufr, IMG_SIZE );
   fprintf(stderr, "Read image\n");
   img_id = ad_allocate_image(HEIGHT, WIDTH, sizeof(unsigned char));
   res_id = ad_allocate_image(HEIGHT, WIDTH, sizeof(unsigned char));
   fprintf(stderr, "Allocated image\n");
   ad_distribute_image( img_bufr, HEIGHT, WIDTH, sizeof(char), res_id );
   for (frame = 0;frame<FRAMES;++frame)
     {
     addc1b( res_id, 1, res_id, HEIGHT, WIDTH );
     ad_collect_image_port(out0, res_id);
     }
fprintf(stderr,"master.c.add_one_bw is done...%3d\n", frame);   fflush(stderr);
Terminate_Adapt();
exit(0);
}
/* FUNCTION: read_input -- Read from stdin to the SIB = cell 64 */
read_input(fd, buffer, nbytes)
  int fd;
  char *buffer;
  int nbytes;
{
int nread;
 while((nread = read(fd, buffer, nbytes)) < nbytes) {
 if (nread == 0) {
    fprintf(stderr, "Premature EOF on read!!\n");
    return(-1);
    }
```

```
 buffer += nread;
 nbytes -= nread;
 }
 return(0);
}
/* FUNCTION: write_output -- Write to stdout to the SIB = cell 64 */
write_output(fd, buffer, nbytes)
  int fd;
  char *buffer;
  int nbytes;
{
while(nbytes>MAXSIZE)
  {
  if (write(fd, buffer, MAXSIZE) != MAXSIZE)
    {
    fprintf(stderr, "Couldn't complete write!\n");
    return(-1);
    }
  nbytes -= MAXSIZE;
  buffer += MAXSIZE;
  }
if (write(fd, buffer, nbytes) != nbytes)
  {
  fprintf(stderr, "Couldn't complete write!\n");
  return(-1);
  }
return(0);
}
```

```
/* FILE: ~symanski/iwarp/documents/report/frame.c.add_one_bw
*
*    Uses stdin0064 for 512x512 image.
*
*    frame.c from Jon Webb with additions by Symanski to drive the Sony monitor.
*
*    This program writes frames at about 18 FPS to a 512x512 window centered
*    in the the 1kx1k screen. It uses the ib_receive_words which does a single
*    word write to FCSW to guarantee an addressing transition.  Writes take
*    about 650 nanoseconds each.
*
*/
#include <stdio.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pcs/pcs_def.h>
#include <iwsys/getcfg.h>
#include <netcode.h>
#include <espl.h>
#include <malloc.h>
#include <fastio.h>
#include <pcs/pcs_time.h>
#include "ib.h"              /* symanski's image board library file */
#define HEIGHT 512
#define WIDTH  512
#define FRAMES 1000000
#define PCS_ERROR { fprintf(stderr, "PCS Error in file %s line %d\n", \
__FILE__,__LINE__);     pcs_fatal(NULL); }
main() {
  int check, line, offset;
  char  *image = (char *) malloc(HEIGHT * WIDTH * sizeof(unsigned char));
  int in_port, frame=0, adr=0;
ENABLE_IMAGE_BOARD();    /* initialize image board */
DISABLE_PARITY();        /* load null parity handler */
LOAD_8BPP();             /* setup graphics chip - grey LUT */
COLOR_DISPLAY8(200);
/*RAMP();                   display a ramp pattern */
/*LOAD_LUT8();              load spectrum LUT */
/*LOAD_CHECKERS8( 255 );    display a checker pattern */
offset = (256 * 256) + 63;
  pcs_init(ports, NUM_PORTS(ports), NULL, 0);
  bind_systolic_gate(in0, GATE0);
  in_port = esplc_bind_receive_port(in0);
  fprintf(stderr, "frame.c.add_one_bw: \nStarting frames\n"); fflush(stderr);
  for (frame=0; frame<FRAMES; frame++)
    {
    if (!receive_open_msg(in0, GATE0)) PCS_ERROR;
    for ( line=0; line<512; line++ )
      {
      adr = (line * 256);
      ib_receive_words( 128, &VRAM[adr + offset ], FCSW ); /* single write only */
      }
    FCSW[0] = 0x08;  /* Turn  LED on - Used to check hang point */
    if (!receive_close_msg(in0, GATE0)) PCS_ERROR;
    if ( frame%1000 == 0)  fprintf(stderr, "Frame: %6d\n", frame); fflush(stderr);
    }
  fprintf(stderr, "frame.c.add_one_bw is done...  %d\n", frame );  fflush(stderr);
  exit(0);
}
```

```
--   FILE: ~symanski/iwarp/documents/report/frame.ad.add_one_bw
--
--  This file must contain ALL adapt functions used in master.c
--
procedure addclb(image1    : in image byte,
                     constant : in integer,
                     image2   : out image byte)
is
next begin
    image2 := image1 + constant;
end next;
end addclb;
procedure setvalues(  im : out image byte,
                     val : in integer     )
is
next begin
    im := val;
end next;
end setvalues;
--  add this to test for no frame call
--
-- add_one.ad -- do simple operation on a byte
--       image (put this in the add_one.ad file)
procedure add_one(img_in  : in  image byte,
                     img_out : out image byte )
IS
FIRST BEGIN          -- no initialization
 ;
END FIRST;
NEXT BEGIN           -- add one to each pixel
         img_out := img_in + 64;  -- add 1 to pixel
END NEXT;
COMBINE BEGIN        -- combine image
  img_out := img_out +  _img_out;
END COMBINE;
END add_one;
--  scroll function from Jon Webb - 12 Aug 1993
--
procedure scroll_left(inimg  : in  image array(-1..1,-4..4) of byte border 128,
                      outimg : out image byte)
is
next begin
  outimg := inimg(0, 4); -- outimg has the pixel from one column to the RIGHT
end next;                --   so the image moves LEFT
end scroll_left;
--  scroll_up function from Jon Webb - 12 Aug 1993
--  Modified to scroll down by symanski
--
procedure scroll_down(inimg : in  image array(-1..1,-1..1) of byte border 128,
                 outimg: out image byte)
is
next begin
  outimg := inimg(-1,0); -- outimg has the pixel from one row BELOW the input
end next;                -- so the image moves down
end scroll_down;

-- load image from one buffer to another
--
procedure load_img(imag      : in image byte,
```

```
                        constant : in integer,
                        image2   : out image byte)
    is
    next begin
        image2 := image1 + constant;
    end next;
    end load_img;
```

```
/*  FILE:  ~symanski/iwarp/documents/report/fastio.h.add_one_bw
*
*   Various assembly code functions for adapt.
*
*/
#define BEGINCA {
#define ENDCA }
/*   This works but is a single word write  -
     Note st.f cnt,(b) line to write to CS reg */
asm void ib_receive_words(n,a,b) {
% tmpreg n,a,b,cnt; use ga0; lab less4,again,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  .beginloop .LOOP
  .c005 12
  loop n
  .c186_loop_okay
  st.f cnt,(b)
el   st.f ga0,(a,4)+=
  .endloop .LOOP
finish:
  nop
}
asm void copy_words(n,r,w) {
% tmpreg n,r,w,cnt; use ga0,lm0,lm1,lm2,lm3,lm4; lab extra,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  sub 1,n
  ld (r,4)+=,lmw
  ld (r,4)+=,lmr1
  .beginloop .LOOP
  .c005 16
  loop n
  .c186_loop_okay
el BEGINCA fmova lmr1,lmw; ld (r,4)+=,lmr1; st lmw,(w,4)+= ENDCA
  .endloop .LOOP1
  st lmw,(w,4)+=
  st lmr1,(w,4)+=
finish:
  nop
}
asm void copy_for_transpose(m,n,r,incr1,incr2,w,incw1,incw2) {
  % tmpreg m,r,w,d; register n,incr1,incr2,incw1,incw2; use lm0,lm1,lm2,lm3,lm4; lab loop1,
finish;
loop1:
  .beginloop .LOOP
  loop n
    ld.b (r,incr2)+=,d
el   st.b d,(w,incw2)+=
  .endloop .LOOP
  add incr1,r
  add incw1,w
flags sub 1,m
  brifn ilu.zero,loop1
}
asm void copy_words_for_transpose(m,n,r,incr1,incr2,w,incw1,incw2) {
  % tmpreg m,r,w,d; register n,incr1,incr2,incw1,incw2; use lm0,lm1,lm2,lm3,lm4; lab loop1,
```

```
finish;
loop1:
  .beginloop .LOOP
  loop n
    ld (r,incr2)+=,d
el  st d,(w,incw2)+=
  .endloop .LOOP
  add incr1,r
  add incw1,w
flags sub 1,m
  brifn ilu.zero,loop1
}
asm void pass_words(n) {
% tmpreg n,cnt; use ga0; lab less4,again,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  .beginloop .LOOP
  .c005 8
  loop n
  .c186_loop_okay
el  movereg ga0,ga0
  .endloop .LOOP
finish:
   nop
}
asm void receive_words(n,a) {
% tmpreg n,a,cnt; use ga0; lab less4,again,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  .beginloop .LOOP
  .c005 8
  loop n
  .c186_loop_okay
el  st.f ga0,(a,4)+=
  .endloop .LOOP
finish:
  nop
}
asm void send_words(n,a) {
% tmpreg n,a,cnt; use ga0; lab less4,again,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  .beginloop .LOOP
  .c005 8
  loop n
  .c186_loop_okay
el  ld.f (a,4)+=,ga0
  .endloop .LOOP
finish:
  nop
}
asm void receive_pass_words(n,a) {
% tmpreg n,a,cnt; use ga0; lab finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
```

```
  .beginloop .LOOP
  .c005 16
  loop n
  .c186_loop_okay
el  BEGINCA fmovm ga0,ga0; st.f ga0,(a,4)+= ENDCA
  .endloop .LOOP
finish:
  nop
}
asm void receive_pass2_words(n,a) {
% tmpreg n,a,cnt; use ga0,ga2; lab less4,again,finish;
  clr cnt
  cmp cnt,n
  brif ilu.zero,finish
  .beginloop .LOOP
  .c005 16
  loop n
  .c186_loop_okay
el  BEGINCA fmovm ga0,ga0; fmova ga0,ga2; st.f ga0,(a,4)+= ENDCA
  .endloop .LOOP

finish:
  nop
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Final: March 1991–September 1993

4. TITLE AND SUBTITLE

iWARP DISPLAY MODULE

5. FUNDING NUMBERS

602234N
CS2D
RS34J77

6. AUTHOR(S)

J. J. Symanski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division
San Diego, CA 92152–5000

8. PERFORMING ORGANIZATION REPORT NUMBER

TD 2610

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
800 North Quincy St.
Arlington, VA 22217–5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes the design, fabrication, and testing of the iWARP display module. The display module is a custom circuit board designed specifically for the Intel iWarp processor. Software written during this development enables the board to generate video signals to drive a high-resolution color display, with images processed within the iWarp processor array.

The module contains 4 megabytes of VRAM which will hold images of user-determined pixel depth and size. Image data are converted to analog video signals by the Inmos G364 color video controller chip. Image sizes can range from 1024 by 1024 24-bits-per-pixel true-color images to 1-bit-per-pixel monochrome images.

As part of this development, several iWarp programs have been developed to aid in the use of the display. To the image processing application developer, data can be displayed with a simple one-line subroutine call.

14. SUBJECT TERMS

high-resolution displays
iWARP

15. NUMBER OF PAGES

79

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE (Include Area Code)	21c. OFFICE SYMBOL
J. J. Symanski	(619) 553–2530	Code 761

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0274B	Library	(2)
Code 0275	Archive/Stock	(6)
Code 402	R. A. Wasilausky	(1)
Code 70	T. F. Ball	(1)
Code 76	J. R. Wangler	(1)
Code 7601	K. N. Bromley	(1)
Code 761	G. W. Byram	(1)
Code 761	J. J. Symanski	(10)

Defense Technical Information Center
Alexandria, VA 22304-6145 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363-5100

Center for Naval Analyses
Alexandria, VA 22302-0268

Navy Acquisition, Research and Development
Information Center (NARDIC)
Arlington, VA 22244-5114

GIDEP Operations Center
Corona, CA 91718-8000

NCCOSC Division Detachment
Warminster, PA 18974-5000

Office of Naval Research
Arlington, VA 22217-5660

Carnegie Mellon University
Pittsburgh, PA 15213

Approved for public release; distribution is unlimited.